UNIVERSIDAD DE LA RIOJA

TRABAJO FIN DE ESTUDIOS

Título

Comparación de técnicas deep y tradicionales para la detección de objetos

Autor/es

ALEJANDRO MARTÍNEZ MARTÍNEZ

Director/es

JÓNATAN HERAS VICENTE y ÁNGELA CASADO GARCÍA ,

Facultad

Facultad de Ciencia y Tecnología

Titulación

Grado en Ingeniería Informática

Departamento

MATEMÁTICAS Y COMPUTACIÓN

Curso académico

2018-19

Comparación de técnicas deep y tradicionales para la detección de objetos, de ALEJANDRO MARTÍNEZ MARTÍNEZ (publicada por la Universidad de La Rioja) se difunde bajo una Licencia Creative Commons Reconocimiento-NoComercial-SinObraDerivada 3.0 Unported. Permisos que vayan más allá de lo cubierto por esta licencia pueden solicitarse a los titulares del copyright.

© El autor, 2019
© Universidad de La Rioja, 2019
publicaciones.unirioja.es
E-mail: publicaciones@unirioja.es

UNIVERSIDAD DE LA RIOJA

Facultad de Ciencia y Tecnología

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Comparación de técnicas deep y tradicionales para la detección de objetos

Realizado por:

Alejandro Martínez Martínez

Tutelado por:

Ángela Casado García

Jónathan Heras Vicente

Logroño, junio, 2019

Agradecimientos

Agradezco en primer lugar a mis tutores Ángela y Jónathan su paciencia y dedicación con mi trabajo durante estos meses. También agradezco al equipo al completo de la empresa Pixelabs las facilidades que me han dado para sentirme a gusto en la empresa, y sobre todo a mis compañeros Javi, Cento y Mario que han estado disponibles siempre que lo he necesitado. A mi amigo Germán, que me ha echado una mano cuando lo he necesitado. A mis padres, que me han apoyado siempre. Y a Leire, que me ha ayudado a superar todas las adversidades que han ido surgiendo aun cuando me parecía imposible hacerlo.

Resumen

La detección de objetos en imágenes es una tarea enmarcada dentro de la inteligencia artificial, y más concretamente en la visión por computador. La aparición en estos últimos años de nuevas técnicas basadas en *Deep Learning* ha permitido que el ámbito de la detección cobre una gran importancia gracias a su utilización en múltiples campos como la medicina, la videovigilancia o la detección de logos en vídeos. Sin embargo las técnicas más tradicionales todavía son capaces de presentar resultados competitivos frente a las nuevas alternativas haciendo uso de sistemas más sencillos. Por ello el objetivo de este trabajo es analizar y comparar el rendimiento de técnicas tradicionales y técnicas *deep* en la detección de logos en vídeos.

Abstract

Object detection in images is a task framed within artificial intelligence, and more specifically in computer vision. The rise in recent years of new techniques based on Deep Learning has allowed the field of object detection to become very important thanks to its use in multiple fields such as medical imaging, video surveillance or the detection of logos in videos. However, more traditional techniques are still able to present competitive results against new alternatives using simpler systems. Therefore, the goal of this work is to analyze and compare the performance of traditional techniques and deep techniques in the detection of logos in videos.

Índice

1. Introducción

En este apartado se expone el contexto en el que se enmarca este proyecto. Además, se introducen los conceptos básicos para la comprensión del trabajo, y se define el objetivo del mismo.

1.1 Contexto

La empresa Pixelabs SL es una empresa dedicada a la creación de soluciones basadas en inteligencia artificial. El enfoque de sus proyectos es el de particularizar las distintas soluciones que han implementado para cada cliente, adaptándose a sus necesidades y requisitos. De este modo, se consigue un software capaz de aportar valor al cliente para automatizar procesos u obtener información útil a partir de sus propios datos.

Como norma general, estos proyectos informáticos están compuestos de una parte de presentación en la que se muestra al cliente los datos analíticos generados por su solución, y de un motor, o *backend*, que es el que se encarga de generar los datos apoyándose en visión por computador y redes neuronales.

La tarea más frecuente en estos proyectos es la detección de objetos, que consiste en encontrar la localización de uno o varios objetos en una imagen o vídeo. Es así como en Pixelabs han desarrollado proyectos de detección de logos en videos, o de seguimiento de personas en almacenes.

Sin embargo, dentro de la detección de objetos existen numerosas técnicas distintas que resuelven mejor unos problemas que otros. Es por ello que Pixelabs necesita conocer qué técnicas de detección de objetos son las más adecuadas en función del proyecto al que se apliquen.

Durante las prácticas realizadas en esta empresa, pude familiarizarme con algunas técnicas de detección de objetos y con los procesos derivados de esta tarea.

1.2 Detección de objetos

La visión por computador [1] es el campo de la inteligencia artificial que se ocupa de cómo los ordenadores “ven” imágenes o vídeos. Dentro de este campo se encuentran cuatro grandes tareas: clasificación, localización, detección y segmentación [2].

Clasificación de imágenes

La clasificación de imágenes se basa en asignar una etiqueta a una imagen. Por ejemplo, la Figura 1 es clasificada como gato.

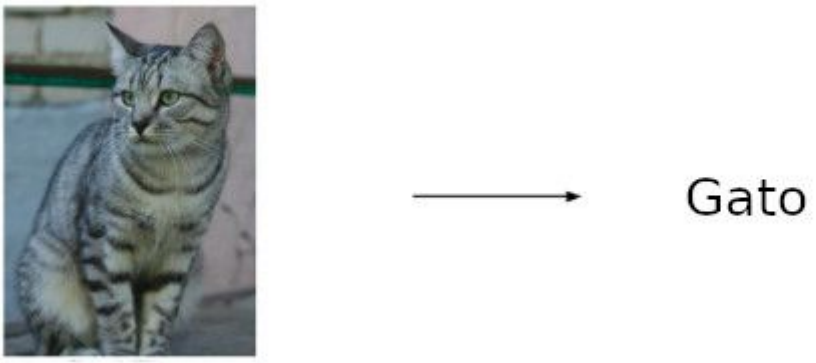


Figura 1. Clasificación de imágenes

Localización de objetos

La localización de objetos es una tarea cuyo fin es encontrar la posición de un objeto en una imagen. En la Figura 2 se puede ver que el objeto principal de la imagen es ubicado dentro del recuadro rojo.

Figura 2. Localización de objetos

Segmentación

La segmentación se basa en realizar el etiquetado de cada píxel en una imagen. Se define por un lado la segmentación semántica (parte izquierda de la Figura 3), en la que se le asigna una categoría a cada píxel, sin diferenciar entre las distintas instancias de la misma clase en la imagen; y por otro lado, la segmentación de instancias en la que sí se discriminan las distintas instancias de la misma clase (parte derecha de la Figura 3).

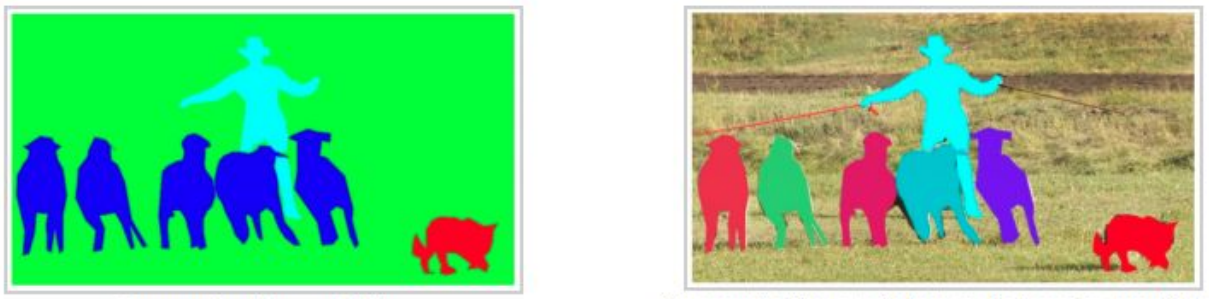


Figura 3. **Izquierda:** segmentación semántica. **Derecha:** segmentación de instancias

Detección de objetos

Finalmente, la detección de objetos consiste en localizar múltiples objetos dentro de una imagen y dar su categoría. En la Figura 4 se puede ver un ejemplo de detección de objetos, donde se han detectado varios coches, camiones y semáforos en la imagen.

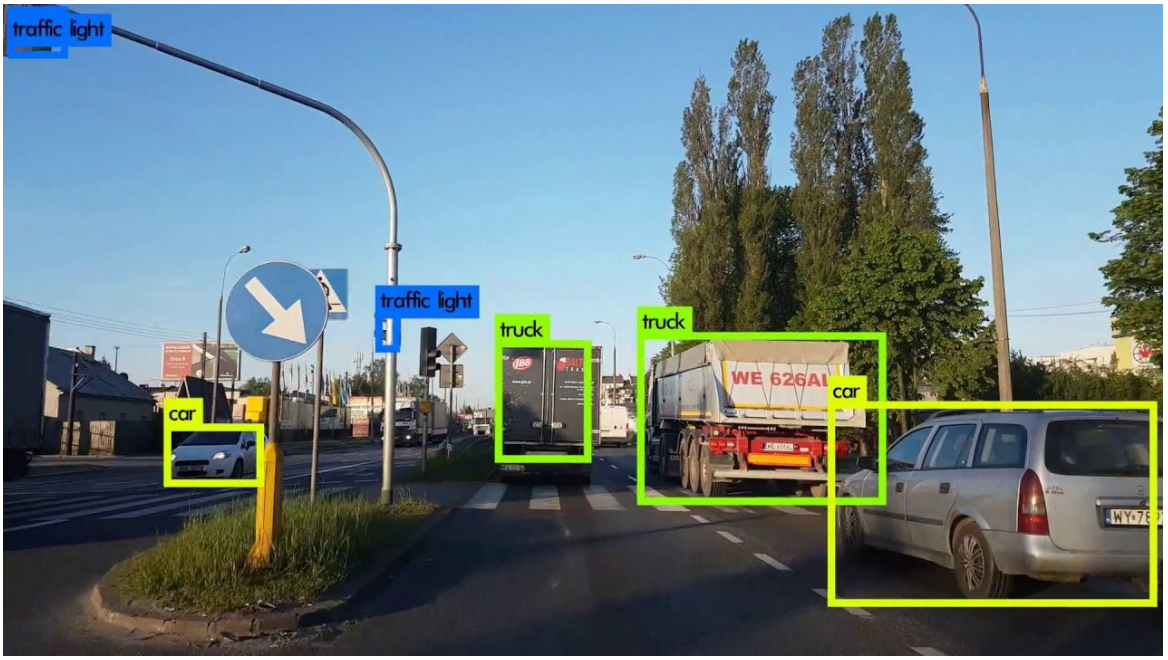


Figura 4. Detección de objetos

La detección es la tarea en la que se centra este proyecto ya que es la más frecuente dentro de la empresa Pixelabs.

El método habitual de resolución de esta tarea es el que se ve en la Figura 5. A partir de un algoritmo y de un grupo de imágenes (en las que manualmente se han indicado la posición de los objetos) se realiza un entrenamiento (1). Como resultado de éste, se obtiene un modelo (2). A este modelo se le introducen datos, en este caso imágenes, para los que genera predicciones (3), que son los recuadros que indican dónde se encuentran los objetos de interés, es decir, los objetos a detectar, en función de los patrones que ha aprendido el modelo durante la fase de entrenamiento.

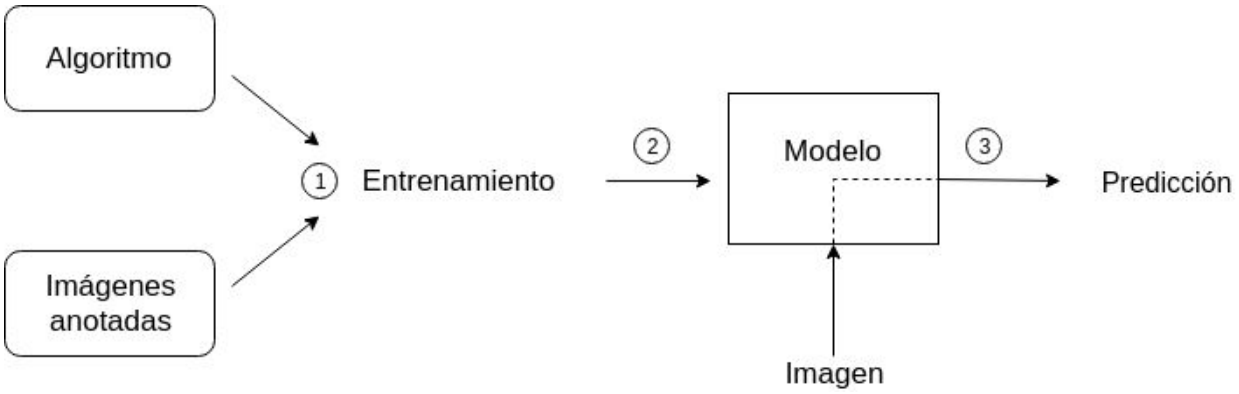


Figura 5. Diagrama de entrenamiento de un modelo y obtención de predicciones

Actualmente existen dos aproximaciones a la hora de enfrentar la construcción de modelos de detección, la tradicional y la moderna basada en técnicas de *deep learning*.

El objetivo de este trabajo será el de comparar ambas aproximaciones para saber cuál se ajusta mejor a las necesidades de un proyecto de detección de logotipos en imágenes. En concreto se compararán distintos modelos para la detección del logo Motul en eventos deportivos, con el objetivo de localizar la posición de este logo en una imagen como se ve en la Figura 6.

Figura 6. Imagen de ejemplo de detección del logo Motul

2. Planificación

En esta sección se presenta la organización y programación de las tareas que componen el proyecto, la metodología a seguir y el plan de riesgos.

2.1 Alcance

El objetivo de este proyecto es el análisis y comparación de técnicas de detección de objetos tradicionales frente a las más modernas basadas en *deep learning*. Dada la gran variedad de técnicas existentes, y orientado desde la empresa y por los tutores, se opta por comparar dos técnicas. En concreto:

- HOG, una técnica tradicional.
- YOLO, una técnica basada en *deep learning*.

2.2 Requisitos funcionales

Los requisitos funcionales para este proyecto son:

- Entrenar un modelo basado en HOG para detectar el logo Motul.
- Entrenar un modelo basado en YOLO para detectar el logo Motul.

2.3 Requisitos no funcionales

Los requisitos no funcionales para este proyecto son:

- Comparar ambos modelos con respecto a su precisión.
- Comparar ambos modelos con respecto a su velocidad de procesado.
- Comparar los tiempos de entrenamiento.

2.4 Ampliaciones

En caso de que durante el desarrollo de este proyecto no surjan problemas notables y su duración se vea acortada, se estudiarán otras técnicas para incluir en la comparación.

2.5 Metodología

La metodología a seguir en este proyecto será la iterativo-incremental. Se ha elegido esta metodología dado el carácter modular propuesto y a que es el procedimiento más adecuado para gestionar iteraciones que van añadiendo información a la comparativa.

Se proponen las siguientes iteraciones:

- Iteración 1: en la que se entrenará un modelo basado en HOG.
- Iteración 2: en la que se entrenará un modelo basado en YOLO.
- Iteración 3: en la que se realizará la comparativa entre ambos modelos.

Las dos primeras iteraciones estarán compuestas por análisis, diseño y entrenamiento. La tercera iteración estará compuesta por el análisis de las métricas para la evaluación y la comparativa entre ambos modelos.

2.6 Calendario e hitos

En la Figura 7 pueden verse los hitos del proyecto en el calendario, y en la Tabla 1 los hitos por semana.

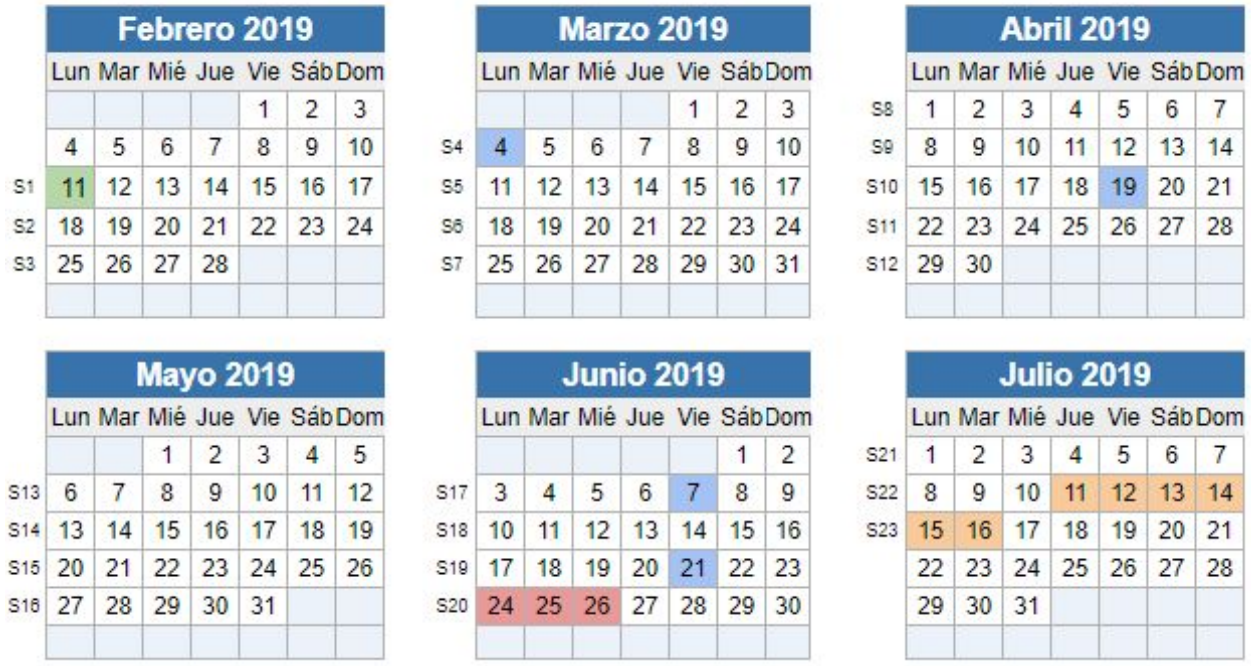


Figura 7. Calendario con las fechas importantes

	Semana																					
	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Comienzo del TFG	♦																						
Fin planificación y comienzo iteración 1				♦																			
Fin iteración 1 y comienzo iteración 2										♦													
Fin de iteración 2 y comienzo iteración 3																	♦						
Fin iteración 3																			♦				
Depósito del TFG																			♦				
Defensa del TFG																						♦	

Tabla 1. Diagrama de hitos

La organización del trabajo consta de la planificación, las dos iteraciones, la comparativa y la preparación de la defensa del TFG. La duración de la planificación se ha estimado que será de tres semanas, mientras que la de cada iteración se ha estimado que será de seis semanas. Las dos últimas semanas se han dejado para la elaboración de la comparativa. Una vez depositado el trabajo se dispondrán de tres semanas para preparar la defensa del TFG.

2.7 EDT

En la Figura 8 se muestra la estructura de descomposición de trabajo del proyecto.

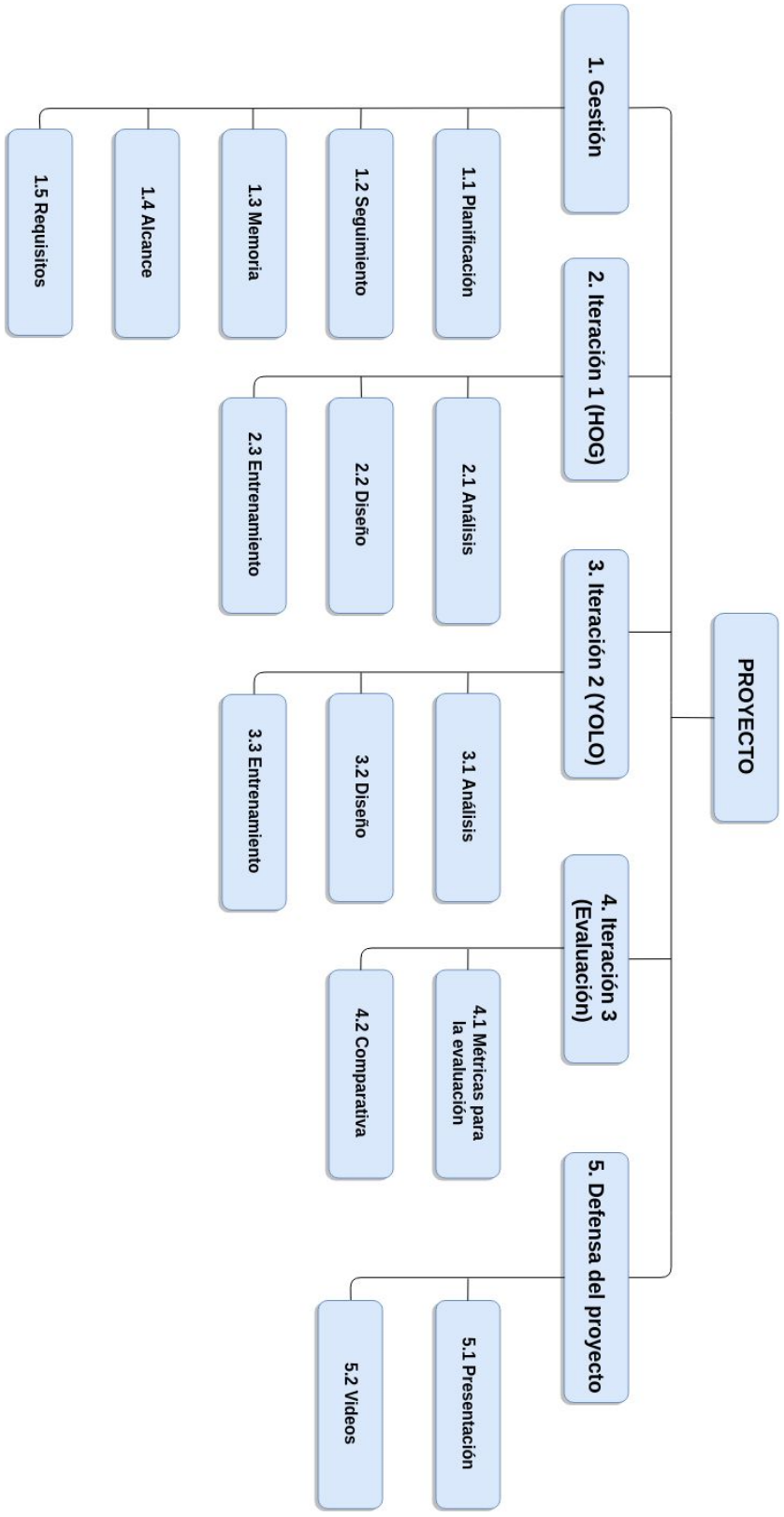


Figura 8. Estructura de descomposición de trabajo

2.8 Diccionario de la EDT

En la Tabla 2 encontramos la descripción de las tareas de la EDT y sus horas previstas.

ID	Tarea	Descripción	Horas previstas
1	Gestión	Conjunto de tareas transversales del proyecto con el fin de controlar los plazos y dejar constancia de los pasos seguidos.	80
1.1	Planificación	Elaboración de un plan de trabajo y de organización del tiempo disponible para el desarrollo del proyecto.	35
1.2	Seguimiento	Control de los plazos establecidos e inclusión de los cambios que se den en el plan de tiempos.	10
1.3	Memoria	Documento redactado conforme va avanzando el proyecto y en el que se plasmarán por escrito las distintas fases de este.	25
1.4	Alcance	Definición del recorrido total que tendrá el proyecto.	5
1.5	Requisitos	Obtención de los requisitos.	5
2	Iteración 1 (HOG)	Conjunto de tareas orientadas al estudio y entrenamiento de un modelo basado en HOG.	90
2.1	Análisis	Estudio de la técnica HOG.	25
2.2	Diseño	Diseño del método y de los pasos que van a usarse para entrenar el modelo basado en HOG.	55
2.3	Entrenamiento	Entrenamiento del modelo.	10
3	Iteración 2 (YOLO)	Conjunto de tareas orientadas al estudio y entrenamiento de un modelo basado en YOLO.	90
3.1	Análisis	Estudio de la técnica YOLO.	25
3.2	Diseño	Diseño del método y de los pasos que van a usarse para entrenar el modelo basado en YOLO.	55
3.3	Entrenamiento	Entrenamiento del modelo.	10
4	Iteración 3 (Evaluación)	Fase de entrenamiento de los modelos.	25
4.1	Métricas para la evaluación	Estudio y elección de las métricas que se van a utilizar para comparar modelos.	15
4.2	Comparativa	Análisis basado en las métricas definidas anteriormente para comparar ambos modelos.	10
5	Defensa del proyecto	Material destinado a la defensa del proyecto ante el tribunal de evaluación.	15
5.1	Presentación	Documento visual para la defensa del trabajo.	14
5.2	Vídeos	Aplicación de los modelos de detección sobre un vídeo de ejemplo.	1

Tabla 2. Descripción y horas de las tareas de la EDT

2.9 Diagrama de Gantt

En la Tabla 3 podemos ver el diagrama de Gantt del proyecto representando las fechas y las duraciones de las distintas tareas.

ID	Tarea	Inicio	Fin	Duración	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17	S18	S19	S20	S21	S22	S23
1	Gestión	11/02/19	21/06/19	80																							
1.1	Planificación	11/02/19	04/02/19	35																							
1.2	Seguimiento	11/02/19	21/06/19	10																							
1.3	Memoria	11/02/19	21/06/19	25																							
1.4	Alcance	11/02/19	04/03/19	5																							
1.5	Requisitos	11/02/19	04/03/19	5																							
2	Iteración 1 (HOG)	04/03/19	19/04/19	90																							
2.1	Análisis	04/03/19	15/03/19	25																							
2.2	Diseño	15/03/19	12/04/19	55																							
2.3	Entrenamiento	12/04/19	19/04/19	10																							
3	Iteración 2 (YOLO)	19/04/19	07/06/19	90																							
3.1	Análisis	19/04/19	03/05/19	25																							
3.2	Diseño	03/05/19	31/05/19	55																							
3.3	Entrenamiento	31/05/19	07/06/19	10																							
4	Iteración 3 (Evaluación)	07/06/19	21/06/19	25																							
4.1	Métricas para la evaluación	07/06/19	14/06/19	15																							
4.2	Comparativa	14/06/19	21/06/19	10																							
5	Defensa del proyecto	21/06/19	11/07/19	15																							
5.1	Presentación	21/06/19	11/07/19	14																							
5.2	Vídeos	21/06/19	11/07/19	1																							

Tabla 3. Diagrama de Gantt

2.10 Plan de riesgos

El primer riesgo de este proyecto radica en la dificultad de la instalación de las librerías necesarias para crear los modelos de detección. De igual manera la obtención de las dependencias para su correcto funcionamiento no es trivial. Es por ello que vamos a seguir un sistema de iteraciones.

Sobre el riesgo derivado de la pérdida de información, optamos por la elaboración de la memoria en la plataforma Google Drive. Además se guardarán copias semanales de la misma en la plataforma Bitbucket, y se guardarán también estas copias en un disco duro particular. Una vez obtenidos los modelos de detección de objetos, se les aplicará esta misma política de copias de seguridad. Como medida adicional, se describirán con detalle los pasos realizados para la obtención de los modelos con el fin de poder replicarlos en caso de una pérdida total de la información.

Los riesgos personales pasan por retrasos debidos a la carga de trabajo que se pueda tener con las demás asignaturas de la carrera. La medida que se tomará para evitar este riesgo será la de conseguir una buena organización del trabajo.

Las comunicaciones con los tutores de la Universidad se mantendrán de forma asíncrona vía correo electrónico, y de forma síncrona mediante reuniones programadas periódicamente. El riesgo de ausencia parcial o incluso total de alguno de los tutores se reduce al contar con dos.

3. Detección de objetos tradicional

En esta sección se analiza una técnica de detección de objetos basada en el método tradicional, se explica su flujo de trabajo y se aplica para construir un modelo de detección.

3.1 Análisis

La detección tradicional de objetos es un método que se basa en el aprendizaje automático [3], el cual es una rama de la inteligencia artificial que permite a los algoritmos aprender a partir de una serie de datos que se les proporcionan durante el proceso de entrenamiento creando modelos de predicción. Gracias a este entrenamiento, los modelos son capaces de generalizar patrones para poder predecir correctamente nuevos datos. Dentro del aprendizaje automático se encuentra el aprendizaje supervisado, que realiza su entrenamiento a partir de pares de objetos: una componente del par son los datos de entrada, y la otra los resultados deseados.

Las técnicas de detección de objetos tradicionales se basan en dividir la imagen en una serie de regiones, y a continuación determinar si en dichas regiones hay, o no, un objeto. El algoritmo encargado de determinar si hay un objeto en una región de la imagen se llama clasificador y es un algoritmo de aprendizaje supervisado.

El proceso de construcción de un modelo de clasificación de imágenes usando aprendizaje supervisado consta de dos pasos. En primer lugar se deben extraer características de la imagen que sirvan para determinar si en ella está o no el objeto, como pueden ser su forma o su textura. Para la extracción de características, los métodos más utilizados son SIFT, SURF [4], las cascadas Haar [5], LBP [6] o HOG [7]. En segundo lugar, a partir de esas características se entrena un algoritmo de clasificación. Entre los algoritmos de clasificación más comunes se encuentran el *Random Forest* [8], KNN [9], SVM [10] o las redes neuronales [11].

Aunque existen múltiples combinaciones de extractores de características y algoritmos de clasificación, una que ha demostrado funcionar correctamente en distintos contextos es la que combina HOG (*Histogram of Oriented Gradients* o Histograma de Gradientes Orientados) con SVM (*Support Vector Machine* o Máquina de Vector Soporte) [12], y esta es la aproximación elegida en este trabajo. Para ello, se va a seguir el esquema de la Figura 9: a partir de un *dataset* de imágenes, es decir un conjunto de imágenes, que contengan o no contengan el objeto de interés, se extraerán los vectores HOG (que se explican a continuación) y con ellos se realizará el entrenamiento del modelo SVM.

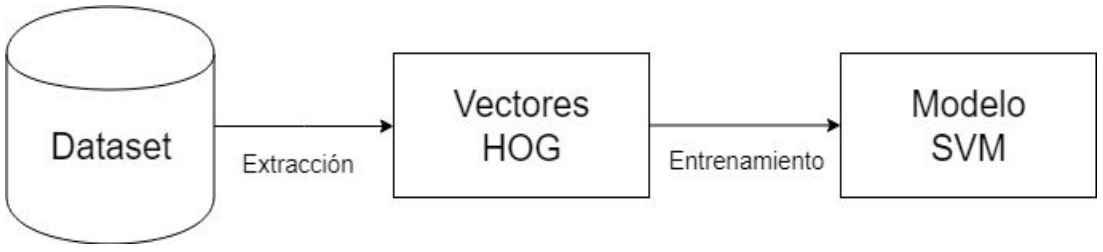


Figura 9. Proceso para construir un clasificador de imágenes combinando HOG y SVM

Como se ha explicado anteriormente, para poder clasificar una imagen en primer lugar hay que obtener las características propias de esa imagen, como pueden ser la forma, el color o la textura. El método de extracción de características HOG presenta la tendencia en la orientación de los gradientes de la imagen.

Se entiende por gradiente el cambio direccional en la intensidad de un píxel. Su valor se define con dos componentes: la dirección hacia la que el cambio de intensidad es máximo (como vemos en la Figura 10), y la magnitud del cambio en esta dirección.

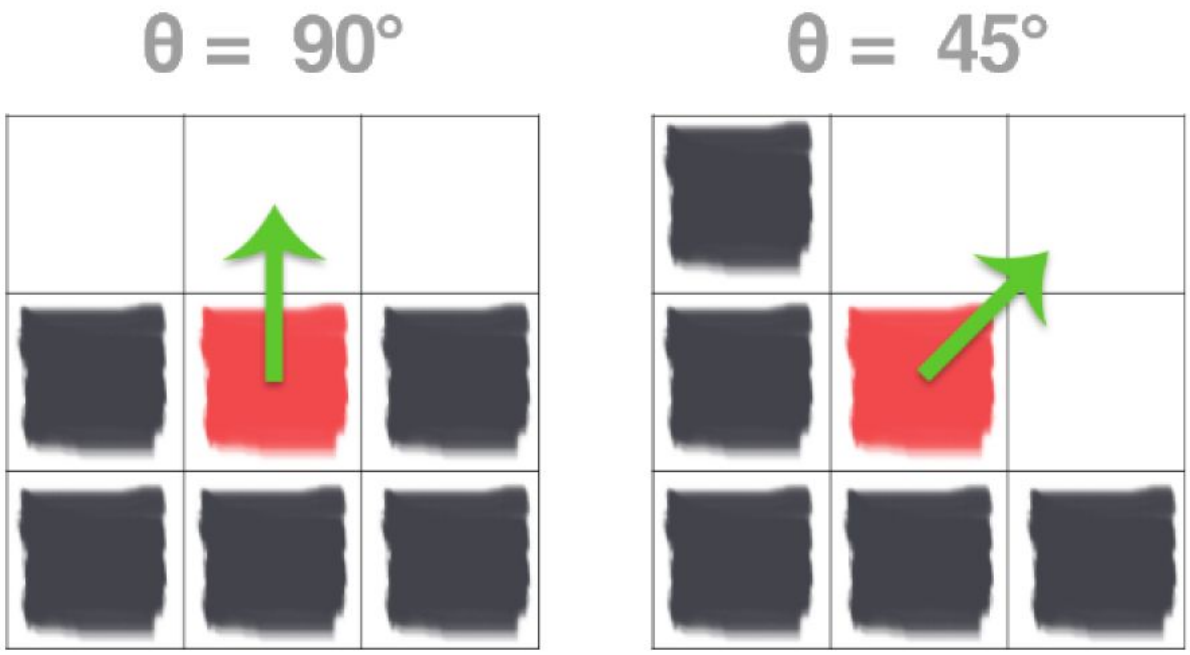


Figura 10. Cálculo del cambio de la orientación desde el píxel central. **Izquierda**: Cambio en orientación desde el píxel central (90º). **Derecha**: En este caso la orientación es más inclinada (45º)

HOG recibe como entrada una imagen, y genera como salida un vector de valores. Cada una de las componentes de este vector contiene la suma de los gradientes de cada píxel de la imagen que tiene su orientación comprendida entre los ángulos de 0º y 20º, 20º y 40º, … , y 160º y 180º respectivamente (un ángulo y su opuesto se consideran iguales en este cálculo) resultando en un total de nueve componentes. En la Figura 11 se puede ver una representación de ese vector en forma de histograma. Cada barra representa una de las componentes, y su tamaño corresponde con el valor de esa componente. Así por ejemplo en la imagen de la que se ha obtenido el histograma de la Figura 11 predominan los gradientes con orientación entre 80º y 100º.

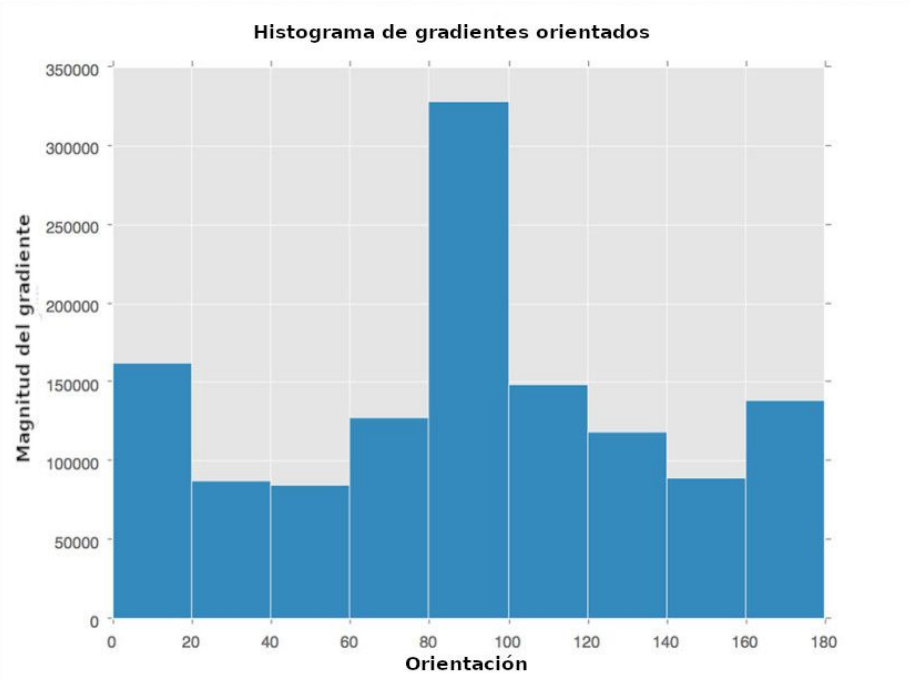


Figura 11. Histograma de Gradientes Orientados

Una vez obtenido el vector con la técnica HOG hay que determinar si la imagen que ha generado este vector contiene el objeto de interés o no. Para ello se entrena un clasificador, en nuestro caso, el algoritmo SVM. Este algoritmo está basado en la idea de encontrar el hiperplano que mejor divida mejor el *dataset* de entrenamiento en dos clases, para así ser capaz de determinar a qué clase de entre las dos pertenece un nuevo dato de entrada. Una vez entrenado el algoritmo, el modelo resultante se puede utilizar para realizar predicciones con nuevas imágenes como se muestra en la Figura 12. Para ello, dada una imagen se extrae su vector de valores usando HOG, se pasa este vector por el clasificador SVM y éste es el que decide si la imagen contiene o no el objeto.

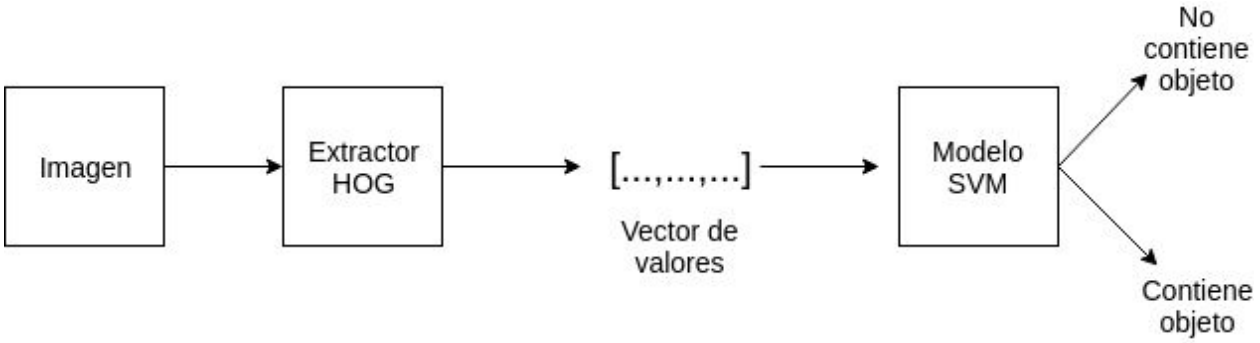


Figura 12. Proceso de clasificación de imágenes combinando HOG y SVM

El flujo de trabajo en la detección de objetos

Una vez elegido el clasificador hay que gestionar cómo se va a dividir la imagen en regiones. Para ello se van a necesitar 3 componentes básicos los cuales son comunes para cualquier

detector basado en clasificación: la ventana deslizante, la pirámide de imágenes y la técnica *non-maximum suppression* [13].

Ventana deslizante

Una ventana deslizante es una región rectangular de altura y anchura fijas que se desplaza a través de una imagen, de izquierda a derecha y de arriba a abajo, moviéndose cada vez un número fijo de píxeles. Para cada una de estas ventanas se toma la región que comprende, llamada región de interés y se pasa por el clasificador para determinar si la ventana contiene o no el objeto. Sin embargo, puede que el objeto que se quiere detectar no quepa dentro de esta ventana. En la Figura 13 se puede ver como la taza de la izquierda cabe en la ventana, mientras que la de la derecha no. Como solución a este problema se introduce la pirámide de imágenes.

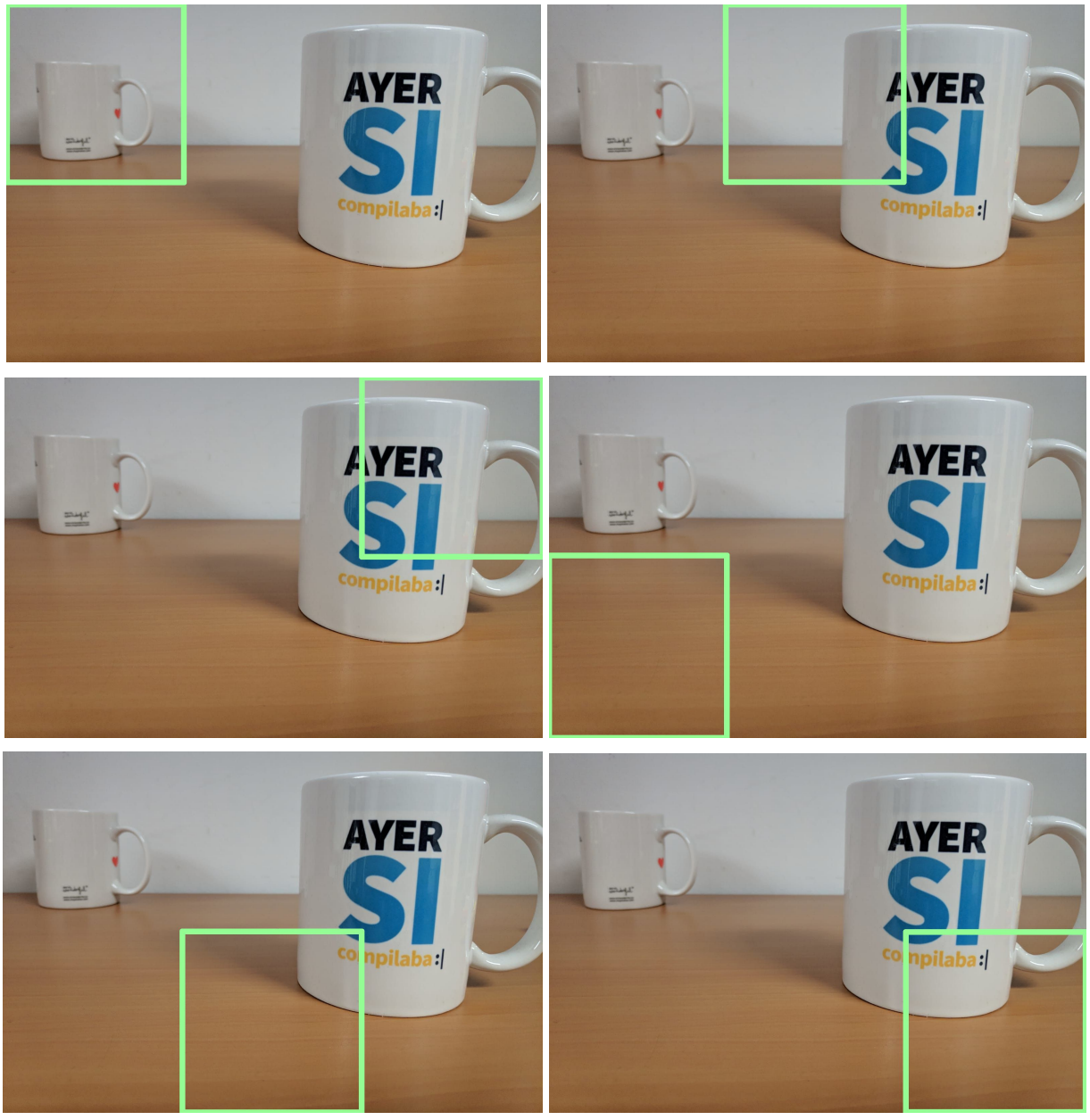


Figura 13. Ventana deslizante

Pirámide de imágenes

La pirámide de imágenes es una representación multiescala de una imagen. En cada capa de la pirámide, la imagen es redimensionada hasta que se llega a un criterio de parada que suele ser un tamaño mínimo, como el tamaño de la ventana. En la Figura 14 se puede ver como tras redimensionar la imagen original la taza de la derecha cabe dentro de la ventana deslizante. Al combinar la ventana deslizante con la pirámide de imágenes se pueden encontrar objetos en imágenes en distintas localizaciones y a distintas escalas.

Figura 14. Pirámide de imágenes

Una vez aplicados estos dos métodos (ventana deslizante y pirámide de imágenes), cuando la ventana se acerca al objeto y empieza a "verlo" marca esas regiones como contenedoras del objeto. Es por ello que se pueden generar múltiples rectángulos delimitadores (que son las cajas que describen la localización de la detección) para un mismo objeto como se ve en la parte izquierda de la Figura 15. Para solucionar este problema, se aplica la técnica *non-maximum suppression*.

Non-maximum suppression

La técnica *non-maximum suppression* funciona calculando la proporción de superposición entre detecciones, y eliminando las que tengan una proporción elevada para eliminar de esta forma repeticiones.

En la Figura 15 se puede ver cómo mediante la ventana deslizante y la pirámide de objetos se obtienen una serie de detecciones de una cara en la imagen de entrada (izquierda). Mediante la técnica *non-maximum suppression* queda determinado que la detección óptima es la que aparece en la derecha de la Figura 15.

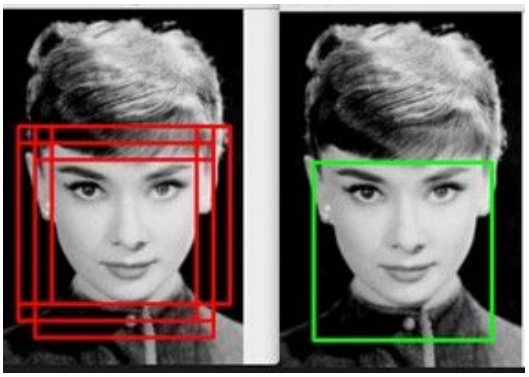


Figura 15. *Non-maximum suppression*

3.2 Diseño

Una vez analizadas las técnicas necesarias para realizar la detección de objetos se pasa a diseñar el proceso de entrenamiento de un modelo. Este proceso consta de la obtención de un *dataset*, su anotación, la partición del *dataset* en los distintos conjuntos (entrenamiento, validación y test), la implementación del algoritmo basado en HOG y SVM, el uso del conjunto de entrenamiento para crear el modelo y el uso del conjunto de validación para elegir los mejores parámetros del modelo. Posteriormente se usa el modelo para procesar el *dataset* de test. Además, dependiendo de la forma en la que se implemente el algoritmo (código propio o librería) puede ser necesario preprocesar las anotaciones de los conjuntos de entrenamiento y de validación. En la Figura 16 se puede ver un esquema del proceso.

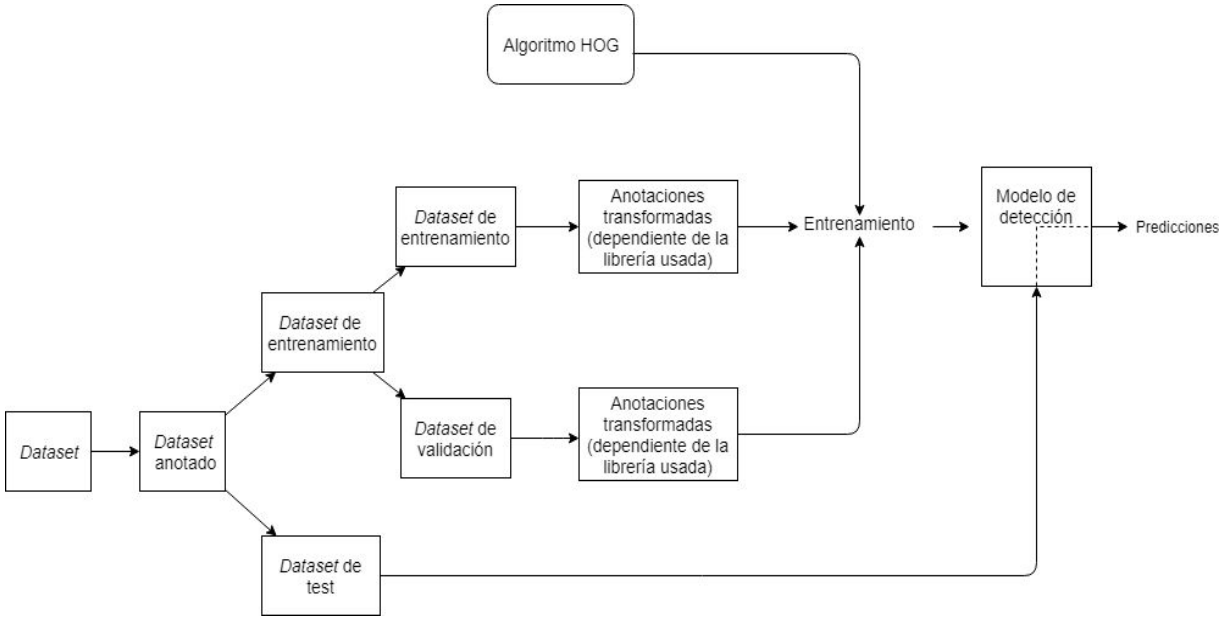


Figura 16. Proceso de creación de un modelo de detección de objetos basado en HOG y SVM

Dataset

En primer lugar, para construir el modelo se va a utilizar un *dataset* de la empresa Pixelabs. Este *dataset* está compuesto por 2500 imágenes de una carrera de MotoGP en las que aparecen algunos logos. Entre otros, el objeto de interés, que es el logo de la marca Motul.

Este paso es independiente del algoritmo o librería que se utilice, ya que en todos los casos se va a necesitar un banco de imágenes para entrenar y realizar las pruebas.

Anotaciones

El segundo paso es el proceso de anotación, que consiste en identificar la posición de los objetos de interés en las imágenes. Este proceso es muy costoso ya que hay que anotar todos los objetos que aparecen en todas las imágenes para evitar confundir al modelo diciéndole que en una parte de la imagen no se encuentra el objeto de interés cuando en realidad sí está. En el caso del *dataset* que se va a usar este proceso ya había sido realizado, y cada imagen del *dataset* cuenta con un XML en el que están anotadas todos los objetos de la imagen asociada. En este archivo cada objeto está definido por sus coordenadas x e y mínimas, que corresponden con la esquina superior izquierda del rectángulo que contiene el objeto, y sus coordenadas x e y máximas, que corresponden con la esquina inferior derecha. En la Figura 17 se puede ver una de las imágenes del *dataset* cruzada con su XML correspondiente, de forma que los logotipos de la marca Motul aparecen destacados por rectángulos.

Figura 17. Imagen con los rectángulos que contienen el logo de Motul

Es importante tener en cuenta que los logos que no aparecen completos no son marcados como detecciones, y por tanto el modelo no debería identificarlos como objeto de interés.

Partición del *dataset*

Una vez anotado, el *dataset* se debe dividir en tres subconjuntos independientes. El primero será el conjunto de entrenamiento. Como su nombre indica, éste servirá para entrenar el modelo. El segundo conjunto será el de validación, el cual se usará para ajustar los parámetros que se pueden configurar a la hora de entrenar el modelo, como son el rigor del clasificador SVM o el tamaño de la ventana deslizante. En último lugar se creará el conjunto de testeo, que es el que permitirá comprobar cómo de bien funciona el modelo. Es importante que los tres conjuntos sean independientes, ya que si durante la fase de testeo se utilizan las mismas imágenes que se han utilizado para entrenar, se le estaría dando una gran ventaja al modelo. Existen múltiples distribuciones para partir los conjuntos de entrenamiento y de test, pero las

más comunes son, en porcentaje del total para entrenamiento y para testeo respectivamente, 66,6%-33,3%, 75%-25% y 90%-10%. En cuanto al conjunto de validación, normalmente se suele reservar un 10-20% del conjunto de entrenamiento para formarlo. Partiendo del *dataset* de 2500 imágenes, se va a hacer una división de 90% para el conjunto de entrenamiento (2250 imágenes) y el 10% restante para el conjunto de test (250 imágenes). Del conjunto de entrenamiento se van a reservar 250 imágenes (un 11%) para el conjunto de validación.

Librería

Ahora que ya se tienen las imágenes necesarias para entrenar hay que implementar el algoritmo basado en HOG y SVM, así como el procesado de las imágenes mediante la ventana deslizante, la pirámide de imágenes y la técnica *non-maximum suppression*. Para ello hay dos alternativas. Una es elaborar todo el código a mano, y otra es utilizar una librería ya desarrollada para ahorrar de esta manera tiempo. La librería más conocida para la implementación de HOG es la librería dlib [14], que incorpora un conjunto de herramientas para el aprendizaje automático y el análisis de datos, y es la que se va a utilizar. En la documentación de esta librería se especifica la entrada que recibe para entrenar un modelo, y al ser diferente a la anotación de la que se dispone habrá que realizar una transformación.

Transformación

El proceso de transformación de las anotaciones es necesario ya que, como se puede ver en la Figura 18, el formato que usa la librería dlib (parte derecha) es el de un solo XML que contiene los nombres de las imágenes, y para cada una de ellas la posición de los objetos localizados por su x e y mínimas (esquina superior izquierda) y por la anchura y altura del rectángulo. Por tanto, ha sido necesario crear un *script* para realizar esta transformación.

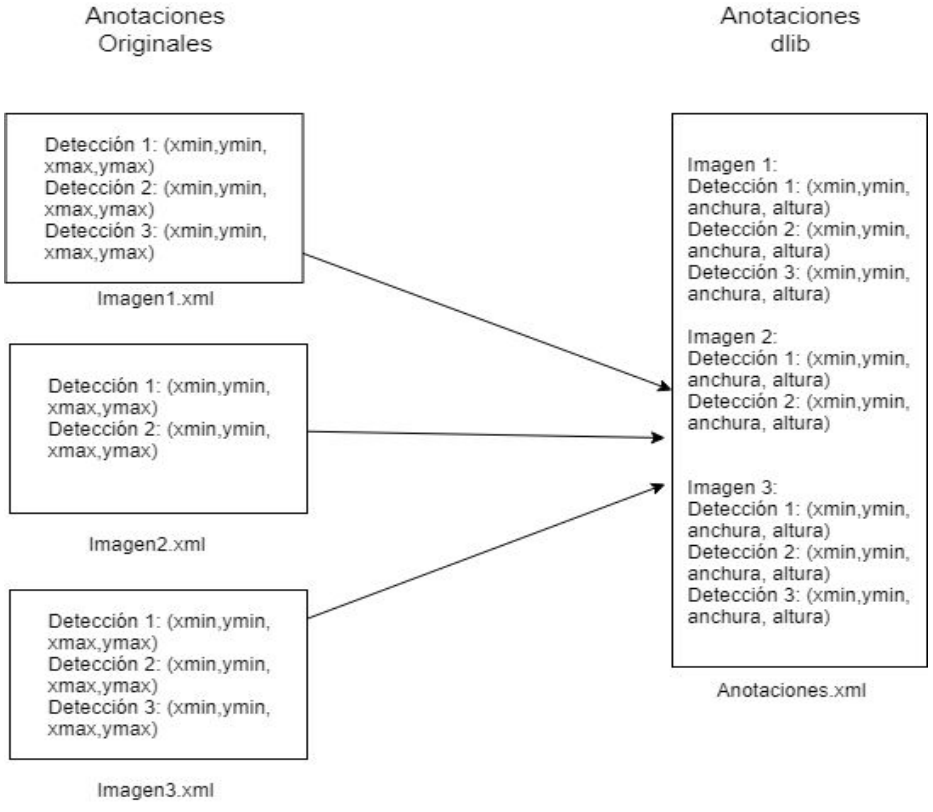


Figura 18. Esquema de transformación de anotaciones

3.3 Entrenamiento

Una vez que se tiene el *dataset* anotado en el formato dlib el siguiente paso consiste en realizar el entrenamiento.

Requisitos

Para poder utilizar la librería dlib los requisitos que el ordenador debe cumplir son:

- Sistema operativo Ubuntu 16.04 o posterior.
- Python. En nuestro caso se usará Python 3.6.
- Numpy (librería de vectores y matrices de Python).

En nuestro caso para el entrenamiento se dispone de un ordenador portátil MSI con Ubuntu 16.04 LTS y con 8GB de memoria RAM, que deberían ser suficientes para crear el modelo.

Parámetros y configuración

La librería dlib permite ajustar una serie de parámetros del entrenamiento. Estos son `num_threads`, `C`, `epsilon`, `be_verbose`, `detection_window_size` y `upsample_limit`.

- El parámetro `num_threads` hace referencia al número de núcleos del procesador que se van a usar para el entrenamiento, que en nuestro caso será de doce.
- Seguidamente encontramos el parámetro `C` que es propio del clasificador SVM. Valores más bajos de este parámetro hacen que el modelo se ajuste mejor a las imágenes del conjunto de entrenamiento, pero puede acabar provocando *overfitting*, que es un sobreajuste con respecto a este conjunto lo cual dificulta que el modelo generalice la detección de objetos en otras imágenes. Inicialmente se deja en 1.
- El parámetro `epsilon` indica el error que va a aceptar el modelo. A valores menores, el modelo será más preciso, pero tardará más tiempo en entrenar. Por defecto está en 0.01.
- El parámetro `be_verbose` con valor `True` hace que la terminal muestre los parámetros que se han utilizado para el entrenamiento e información por cada iteración que se complete. Se entiende que una iteración se completa cada vez que se le muestran todas las imágenes de entrenamiento al algoritmo. El entrenamiento finaliza cuando se reduce el error por debajo del parámetro `epsilon`.
- Encontramos también el parámetro `detection_window_size`, que es el área de la ventana de detección. Se va a usar un valor inicial de 20000 píxeles. En función de la relación entre la anchura y la altura de todas las detecciones, es decir el *aspect ratio*, la librería adecúa el tamaño de la ventana al área que se le ha indicado. En nuestro caso, esta ventana tiene 262 píxeles de anchura y 76 píxeles de altura.
- En último lugar se encuentra el parámetro `upsample_limit`, que indica el número de veces que se aumentan las imágenes cuyas detecciones son demasiado pequeñas o que tienen un *aspect ratio* bastante dispar. Por defecto está en 2.

Una vez entrenado el primer modelo y constatado que todo funciona bien, se aplicará la técnica *GridSearch* para encontrar el que mejores resultados genere. Esta técnica consiste en generar modelos con distintas combinaciones de los parámetros que afecten a su rendimiento. En este

caso se va a probar con `C` y `epsilon` iguales a 0.001, 0.01, 0.1, 1, 10 y 100, y `detection_window_size` igual a 10000, 15000 y 20000.

Métricas

Para entender las métricas que presenta la librería dlib y que permiten evaluar los modelos, conviene definir tres nociones: verdaderos positivos, falsos positivos y falsos negativos. Los verdaderos positivos (*true positives,* TP) son las regiones que ha indicado el modelo y que realmente eran objetos. Los falsos positivos (*false positives,* FP) son las regiones que ha indicado el modelo y que realmente no eran objetos. Y los falsos negativos (*false negatives,* FN) son los objetos que el modelo no ha detectado. En base a estas nociones se definen dos métricas para evaluar los modelos. Estas son la *precision*, que se define como el cociente entre las predicciones que ha acertado y todas las predicciones que ha hecho (Figura 19) y el *recall*, que se define como el cociente entre las predicciones que ha acertado y todos los objetos que debería haber detectado (Figura 20).

$$Precision = \frac{TP}{TP + FP}$$

Figura 19. Fórmula de la métrica *precision*

$$Recall = \frac{TP}{TP + FN}$$

Figura 20. Fórmula de la métrica *recall*

Además también se define la métrica *F1 score* que mide la exactitud del modelo y que se calcula como la media armónica de la *precision* y del *recall* como se ve en la Figura 21. Con esta métrica se busca un balance entre que el modelo acierte con las predicciones que hace y que no se deje objetos por encontrar.

$$F_1 = \left(\frac{\mathrm{recall}^{-1} + \mathrm{precision}^{-1}}{2} \right)^{-1} = 2 \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}}$$

Figura 21. Fórmula de la métrica *F1 Score*

Inicio del entrenamiento

La librería dlib se inicia con la instrucción `python3 dlib_custom/train_detector.py --xml train.xml --detector modelos/detector.svm`. El intérprete de Python, `python3`, ejecuta el código situado en `dlib_custom/train_detector.py`, `--xml` hace referencia al argumento mediante el que se le pasa al programa la ruta del fichero XML con las anotaciones que se han obtenido anteriormente; y `--detector` indica la ruta donde se creará el modelo de detección.

Problemas

A la hora de utilizar esta librería han surgido una serie de problemas.

El primer problema que apareció es que el programa acababa con una salida `killed` como se muestran la Figura 22.

```
stone@P1x3labs:~/TFGAlex/TFGAlex$ python3 dlib_custom/train_detector.py --xml tr
ain.xml --detector modelos/detector.svm
[INFO] training detector...
Training with C: 1
Training with epsilon: 0.01
Training using 4 threads.
Training with sliding window 262 pixels wide by 76 pixels tall.
Upsample images...
Upsample images...
Terminado (killed)
stone@P1x3labs:~/TFGAlex/TFGAlex$
```

Figura 22. Instrucción para el entrenamiento y error

El problema es que el ordenador se queda sin memoria RAM. Esto se debe a que la librería aloja en memoria RAM las imágenes sobre las que hace el proceso de *upsampling*, al cual referencia el parámetro `upsample_limit`. Al trabajar con una ventana deslizante con un tamaño igual a la media entre las detecciones, encuentra bastantes detecciones que por ser o muy anchas o muy altas no consigue procesar, y necesita hacer este proceso. Esto resulta en imágenes muy grandes alojadas en memoria RAM. Para solucionar este problema se pasa a un ordenador con 16GB de RAM, pero nuevamente aparece el mismo error, y nuevamente se debe a la memoria RAM, como se puede ver en la Figura 23 donde el *script* de Python está desbordando la memoria.

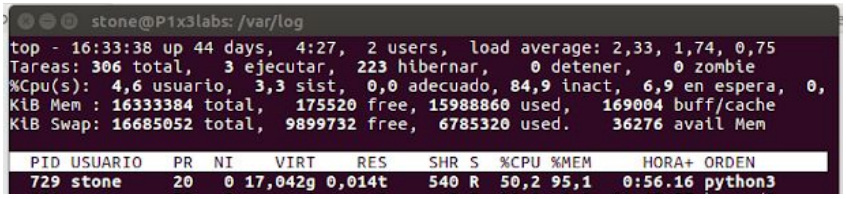


Figura 23. Consumo de memoria RAM del *script* en Python

La manera de solventar este problema es establecer el parámetro `upsample_limit` a 0, ya que de esta forma se bloquea este proceso que es el que está desbordando la memoria RAM. Sin embargo, la consecuencia de este cambio es que directamente no se pueden procesar esas imágenes con detecciones un tanto deformadas, y se produce el error que se ve en la Figura 24, que nos lista las imágenes que en el XML tienen detecciones anómalas.

```
stone@P1x3labs:~/TFGAlex/TFGAlex$ python3 dlib_custom/train_detector.py --xml tr
ain.xml --detector modelos/detector.svm
[INFO] training detector...
Training with C: 1
Training with epsilon: 0.01
Training using 4 threads.
Training with sliding window 262 pixels wide by 76 pixels tall.
Traceback (most recent call last):
  File "dlib_custom/train_detector.py", line 24, in <module>
    dlib.train_simple_object_detector(args["xml"], args["detector"], options)
RuntimeError:
Error! An impossible set of object boxes was given for training. All the boxes
need to have a similar aspect ratio and also not be smaller than about 1250
pixels in area. The following images contain invalid boxes:
  train/motogp_frame25744.jpg
  train/motogp_frame25745.jpg
  train/motogp_frame25746.jpg
  train/motogp_frame25747.jpg
  train/motogp_frame25748.jpg
  train/motogp_frame25749.jpg
  train/motogp_frame25750.jpg
```

Figura 24. Error durante la ejecución del script

Se puede observar que algunas de las detecciones de esas imágenes tienen un *aspect ratio* anormal o un tamaño muy inferior a las demás. En la Figura 25 aparece un ejemplo de estas detecciones anormales. Arriba varias detección con un tamaño (*width* x *height*) menor que 1250, y abajo la representación de las detecciones en la imagen, donde se ve que varias de ellas son mucho más pequeñas.

```
930         <image file="train/motogp_frame25744.jpg">
931                 <box height="23" left="712" top="305" width="80" />
932                 <box height="23" left="540" top="304" width="77" />
933                 <box height="24" left="443" top="303" width="85" />
934                 <box height="22" left="797" top="306" width="75" />
935                 <box height="33" left="624" top="301" width="83" />
936                 <box height="19" left="380" top="311" width="56" />
937                 <box height="18" left="703" top="282" width="63" />
938                 <box height="19" left="325" top="310" width="53" />
939                 <box height="19" left="273" top="310" width="48" />
940         </image>
```

Figura 25. Detecciones anómalas marcadas con rectángulos blancos

Para poder continuar con el entrenamiento hay que preprocesar estas detecciones aumentándoles la altura o la anchura hasta que se acerquen al *aspect ratio* medio, ya que si no la librería dlib no las puede procesar. La repercusión que tiene este preprocesamiento es que esas detecciones no están perfectamente ajustadas al objeto, y ahora incluyen un poco de fondo de la imagen que no corresponde con el objeto. Esto repercutirá negativamente en la precisión del detector al estar empeorando las anotaciones. Tras estos ajustes se reinicia el entrenamiento con el nuevo XML procesado. Esta vez, al haber hecho modificaciones, el tamaño de la ventana deslizante es de 272 píxeles de anchura por 73 píxeles de altura.

Resultados

Tras solventar los problemas anteriores el entrenamiento concluye con éxito. Como se puede ver en la Figura 26, tras 55 iteraciones se genera el modelo y la consola muestra las métricas.

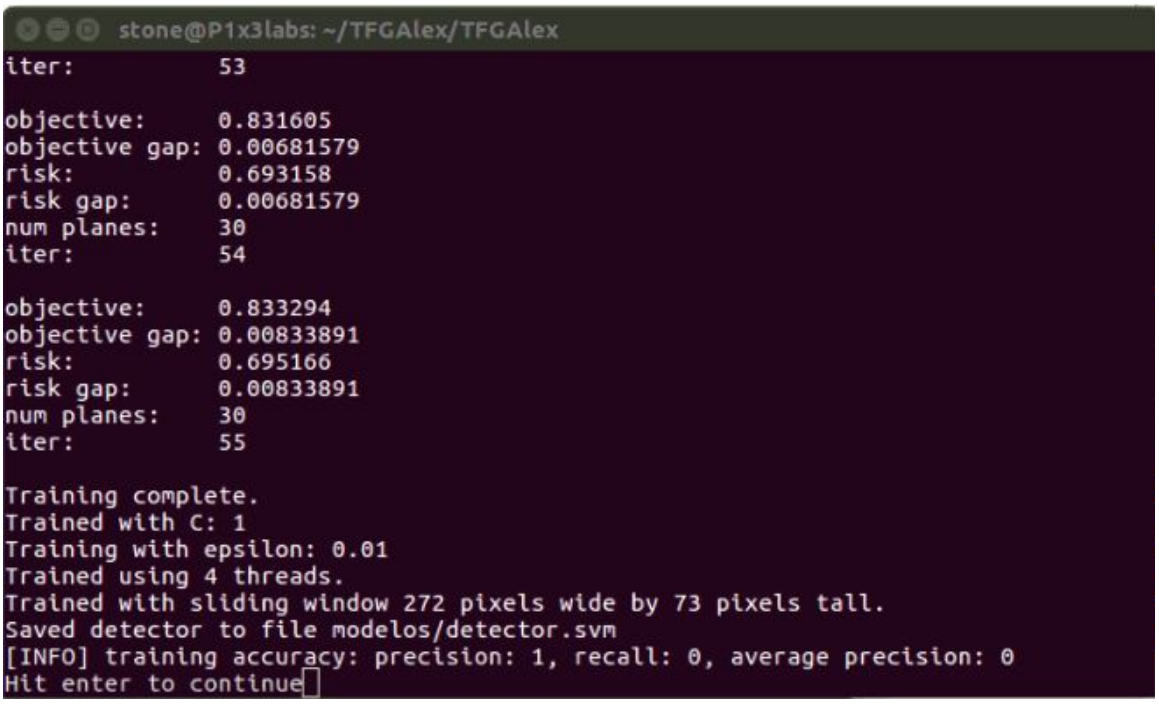


Figura 26. Resultado del primer entrenamiento HOG

Un valor de 1 en la *precision* indica que todas las predicciones que ha hecho sobre el conjunto de test eran correctas, pero el valor de 0 en el *recall* indica que no ha conseguido identificar ningún objeto de entre todos los que había en el *dataset*. Este resultado indica que el modelo no es capaz de encontrar ningún objeto ni siquiera en el conjunto de entrenamiento, en el cual tiene más facilidades para hacerlo ya que ha "visto" todas esas imágenes.

Para encontrar un buen modelo se aplica la técnica *GridSearch* explicada anteriormente. En la Tabla 4 se pueden ver los modelos generados con las combinaciones de estos parámetros. El tiempo medio de cada entrenamiento es de 40 minutos. Sin embargo aparece un comportamiento raro. En muchos modelos se puede ver que se obtienen mejores resultados al procesar el conjunto de validación que al procesar el conjunto de entrenamiento. Esto no es normal dado que los resultados con el conjunto de entrenamiento siempre deben ser mejores, ya que son imágenes que ya ha "visto" el modelo. Tras varios días de investigación se descubre que la razón de este comportamiento reside en el haber puesto el parámetro `upsample_limit` a 0, y en cómo es el *dataset*. Se ha dado la casualidad de que en el conjunto de entrenamiento hay muchas detecciones pequeñas mientras que en el de validación no. Al haber tenido que modificar el parámetro `upsample_limit` esto ha repercutido en que el modelo no ha aprendido bien a detectar estas detecciones pequeñas, y esto ha generado peores resultados en el conjunto de entrenamiento que en el de validación.

Parámetros				Train			Evaluation		
Upsampling Limit	C	Epsilon	Detection Window Size	Precision	Recall	F1 Score	Precision	Recall	F1 Score
0	0,001	100	10000	0,00489345	0,792481	0,009727	0,0337346	0,458419	0,062845
0	0,001	100	15000	0,00762659	0,900752	0,015125	0,0449931	0,450859	0,081821
0	0,001	100	20000	0,00406624	0,559398	0,008074	0,0220775	0,256357	0,040654
0	0,01	100	10000	0,00522329	0,84812	0,010383	0,0379677	0,520275	0,070771
0	0,01	100	15000	0,00774671	0,915789	0,015363	0,0489267	0,493471	0,089027
0	0,01	100	20000	0,00490142	0,673684	0,009732	0,0260201	0,302405	0,047917
0	0,1	100	10000	0,00522329	0,84812	0,010383	0,0379677	0,520275	0,070771
0	0,1	100	15000	0,00774671	0,915789	0,015363	0,0489267	0,493471	0,089027
0	0,1	100	20000	0,00490142	0,673684	0,009732	0,0260201	0,302405	0,047917
0	1	10	10000	0,145417	0,95188	0,252292	0,952839	0,680412	0,793905
0	1	10	15000	0,0429397	0,781955	0,081409	0,003663	0,000687285	0,001157
0	1	10	20000	0,158136	0,933835	0,27047	0,891142	0,573883	0,69816
0	1	100	10000	0,00522329	0,84812	0,010383	0,0379677	0,520275	0,070771
0	1	100	15000	0,00774671	0,915789	0,015363	0,0489267	0,493471	0,089027
0	1	100	20000	0,00490142	0,673684	0,009732	0,0260201	0,302405	0,047917
0	10	0,001	20000	0,962791	0,311278	0,470454	1	0,000687285	0,001374
0	10	0,01	20000	0,964444	0,326316	0,487641	1	0,00137457	0,002745
0	10	0,1	20000	0,963115	0,353383	0,517051	1	0,00206186	0,004115
0	10	1	10000	0,6031785	0,4	0,481014	1	0,038488	0,074123
0	10	1	15000	0,52473	0,866165	0,65354	1	0,17732	0,301227
0	10	1	20000	0,648408	0,765414	0,702069	0,978814	0,158763	0,273211
0	10	10	10000	0,134304	0,971429	0,235982	0,954082	0,771134	0,852908
0	10	10	15000	0,2335227	0,933835	0,373616	0,991004	0,45296	0,62174
0	10	10	20000	0,154319	0,942857	0,265228	0,871166	0,585567	0,70037
0	10	100	10000	0,00522329	0,84812	0,010383	0,0379677	0,520275	0,070771
0	10	100	15000	0,0074671	0,915789	0,014813	0,0489267	0,493471	0,089027
0	10	100	20000	0,00490142	0,673684	0,009732	0,0260201	0,302405	0,047917
0	100	0,001	10000	0,972477	0,478195	0,641129	1	0,0295533	0,05741
0	100	0,001	15000	0,969388	0,857143	0,909817	1	0,109278	0,197025
0	100	0,001	20000	0,971154	0,911278	0,940264	0,976744	0,115464	0,206515
0	100	0,01	10000	0,973294	0,493233	0,654691	1	0,033677	0,06516
0	100	0,01	15000	0,965928	0,852632	0,905751	1	0,108591	0,195908
0	100	0,01	20000	0,972756	0,912782	0,941815	0,974843	0,106529	0,192069
0	100	0,1	10000	0,966102	0,514286	0,671247	1	0,0467354	0,089297
0	100	0,1	15000	0,964974	0,828571	0,891586	1	0,0955326	0,174404
0	100	0,1	20000	0,970874	0,902256	0,935308	0,986577	0,101031	0,183292
0	100	1	10000	0,612162	0,681203	0,64484	0,894273	0,139519	0,241379
0	100	1	15000	0,540395	0,905263	0,676785	0,990769	0,221306	0,361798
0	100	1	20000	0,587576	0,867669	0,700668	0,881679	0,158763	0,269074
0	100	10	10000	0,134359	0,971429	0,236067	0,954082	0,771134	0,852908
0	100	10	15000	0,236206	0,93985	0,37753	0,986686	0,458419	0,625997
0	100	10	20000	0,155045	0,942857	0,266299	0,874615	0,58488	0,700989
0	100	100	10000	0,00522329	0,84812	0,010383	0,0379677	0,520275	0,070771
0	100	100	15000	0,00774671	0,915789	0,015363	0,0489267	0,493417	0,089026
0	100	100	20000	0,00490142	0,673684	0,009732	0,0260201	0,302405	0,047917

Tabla 4. Resultados de los modelos entrenados usando la técnica HOG y dlib

Por tanto se va hacer un rediseño dado que esta forma de implementar la técnica basada en HOG y SVM no sirve en nuestro caso.

3.4 Rediseño

Tras ver que con la librería dlib no se consigue obtener un buen modelo se opta por implementar a mano el código necesario para crear un modelo de detección de objetos siguiendo la técnica HOG y el uso de un clasificador SVM. En este caso no se va a necesitar una transformación de las anotaciones ya que el código va a ser propio y se va a adaptar a las anotaciones que ya se tienen. A diferencia del uso de dlib, se van a obtener recortes positivos (contienen el objeto de interés) y negativos (no contienen el objeto de interés) para pasárselos al clasificador, ya que este proceso lo automatizaba la librería. En la Figura 27 se puede ver un esquema del proceso.

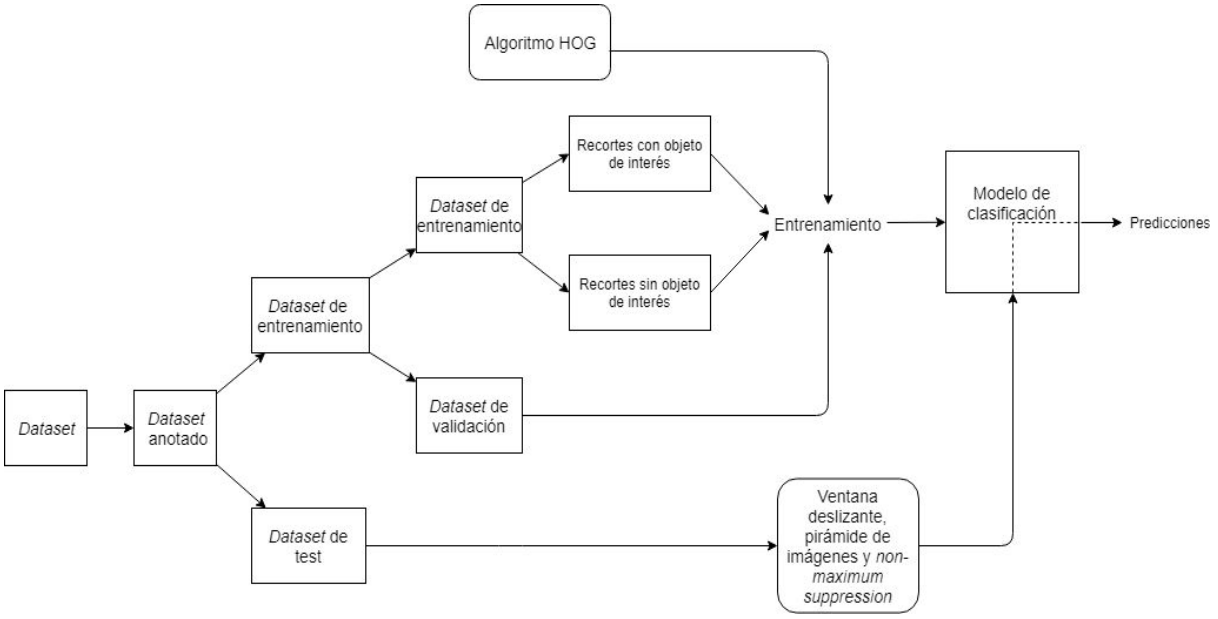


Figura 27. Proceso de creación de un modelo de detección de objetos basado en HOG y SVM con código propio

Tanto el *dataset*, como las anotaciones y la partición del *dataset* son independientes de la técnica que usemos y del algoritmo que implementemos, así que siguen siendo los mismos que al usar la librería dlib.

Implementación

Dados los problemas que han surgido al entrenar los modelos usando la librería dlib se ha optado por implementar a mano el código del entrenamiento. La librería dlib simplificaba el proceso de entrenamiento haciendo automáticamente recortes de las imágenes del conjunto de entrenamiento (unos que contenían el objeto de interés y otros que no) y alojándolos en memoria RAM para luego pasarlos al clasificador SVM. Para evitar los desbordamientos que ocurrían con la librería, se crea un *script* que hace estos recortes y los guarda en disco duro. En el caso de los recortes positivos, son reescalados a un tamaño fijo de 120 píxeles de anchura por 50 píxeles de altura antes de obtener su vector HOG, conservando así el *aspect ratio* ya que todas las detecciones tienen uno similar. En el caso de los negativos, directamente se hace un recorte de esas dimensiones en zonas de las imágenes que no contienen los objetos. Una vez obtenidos todos los recortes se obtiene para cada uno su vector HOG gracias a un *script* propio y este es el dato de entrada que se le pasa al clasificador.

En segundo lugar se crea otro *script* que será el encargado de procesar las nuevas imágenes una vez el modelo ya esté entrenado, haciendo uso de la ventana deslizante, la pirámide de imágenes y de la técnica *non-maximum suppression*.

3.5 Segundo entrenamiento

Una vez elaborados los *scripts* y obtenidos los recortes se puede comenzar el entrenamiento. El entrenamiento se realiza en el mismo ordenador con 16GB de RAM que se usó para realizar el entrenamiento usando la librería dlib.

Requisitos

Dado que el código ha sido desarrollado a mano, los únicos requisitos que deben cumplirse son:

- Sistema operativo Ubuntu 16.04 o posterior.
- Python. En nuestro caso se usará Python 3.6.
- Sklearn (librería para el aprendizaje automático en Python).
- OpenCV >= 2.4 (librería libre de visión por computador). En nuestro caso se usará OpenCV 3.4.0.

Parámetros

Los parámetros de entrenamiento que se van a configurar son `C`, `gamma`, `window_size` y `downscale`:

- El parámetro `C` es el mismo que en el caso de la librería dlib. Inicialmente su valor será de 0.1.
- El parámetro `gamma` hace referencia a cómo de grande es la influencia de una sola imagen de entrenamiento con respecto al modelo total. Inicialmente su valor será de 0.1.
- El parámetro `window_size` es el tamaño de la ventana deslizante. En este caso se dejará un tamaño fijo y será de 120 píxeles de anchura y 50 píxeles de altura para ajustarse a las dimensiones sobre las que se ha entrenado el modelo.
- El parámetro es el coeficiente de reducción de las imágenes en la pirámide de imágenes. Se usará un valor de 1.5.

De nuevo se aplicará la técnica *GridSearch* para encontrar el que mejores resultados genere combinando los valores de los parámetros. El parámetro `C` tomará los valores 0.01, 0.1, 1, 10 y 100, mientras que el parámetro `gamma` tomará los valores 0.001, 0.01, 0.1.

Métricas

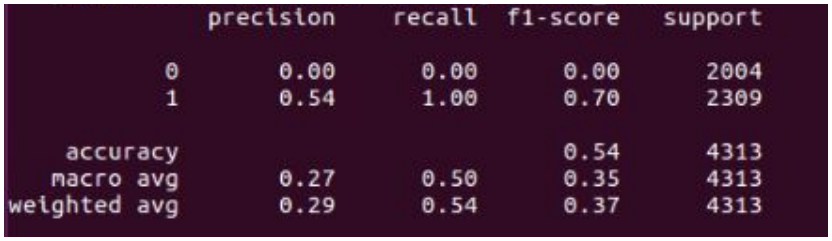
En este entrenamiento se utilizará la métrica *F1 score* obtenida para cada modelo, ya que como se ha visto en el entrenamiento usando la librería dlib abarca las métricas de *precision* y *recall* dándonos un resultado más general.

Inicio del entrenamiento

El programa se inicia invocando el *script* de entrenamiento.

Resultados

Como se ve en la Figura 28, y gracias a la librería sklearn, al finalizar el entrenamiento aparecen las métricas que obtiene el modelo con el conjunto de entrenamiento.



	precision	recall	f1-score	support
0	0.00	0.00	0.00	2004
1	0.54	1.00	0.70	2309
accuracy			0.54	4313
macro avg	0.27	0.50	0.35	4313
weighted avg	0.29	0.54	0.37	4313

Figura 28. Resultado del entrenamiento HOG con código propio

Tras entrenar todos los modelos se han obtenido los resultados que se ven en la Tabla 5. En ella aparecen las métricas registradas para los distintos valores de los parámetros de la técnica HOG implementada a mano sobre los que se ha aplicado la técnica *GridSearch*. El comportamiento es el esperado y a la vista de los resultados se concluye que el mejor modelo se obtiene con `C` igual a 100 y `gamma` igual a 0.001, ya que genera el mejor valor de *F1 score* tanto en el conjunto de entrenamiento como en el de validación.

C	Gamma	F1 score Entrenamiento	F1 score Validación
0.01	0.001	0	0
0.01	0.01	0	0
0.01	0.1	0	0
0.1	0.001	0,09	0
0.1	0.01	0,14	0,02
0.1	0.1	0	0
1	0.001	0,21	0,15
1	0.01	0,37	0,34
1	0.1	0,10	0
10	0.001	0,41	0,13
10	0.01	0,35	0,27
10	0.1	0,27	0,25
100	0.001	0,74	0,65
100	0.01	0,65	0,55
100	0.1	0,65	0,57

Tabla 5. Resultados de los modelos entrenados usando la técnica HOG e implementando el código a mano

Pese a haber tenido que rediseñar el proceso de entrenamiento e implementar el código, el resultado ha sido el de un modelo que esta vez sí concuerda con lo esperado. Su rendimiento en la tarea de la detección del logo Motul es del 65% usando la métrica *F1 score*. Sin embargo las técnicas basadas en *deep learning* que han surgido en los últimos años logran resultados muy competentes en esta tarea, y es por ello que en la siguiente iteración se va a entrenar un modelo basado en una de estas técnicas.

4. Detección de objetos basada en *deep learning*

En esta sección se analiza una técnica de detección de objetos basada en *deep learning*, se explica su flujo de trabajo y se aplica para construir un modelo de detección de objetos.

4.1 Análisis

Las técnicas convencionales de aprendizaje automático trabajan con vectores numéricos, y por lo tanto están limitadas en cuanto a su habilidad para procesar información natural en su forma en bruto (por ejemplo imágenes). Para construir un patrón de reconocimiento o un sistema de aprendizaje automático se requiere de unos conocimientos técnicos sobre el dominio en el que se trabaja para poder construir un extractor de características que transforme esos datos en bruto en una representación adecuada (vector numérico) a partir de la cual el algoritmo pueda aprender. En el caso de la técnica tratada anteriormente en este proyecto, esta representación es el vector HOG que obtenemos de cada imagen.

Frente a esta aproximación tradicional, el aprendizaje profundo o *deep learning* [15] es una clase de algoritmos pertenecientes al ámbito del aprendizaje automático que permite a modelos compuestos por múltiples capas de procesado aprender de manera automática formas de representar los datos (en nuestro caso las imágenes) a través de múltiples niveles de abstracción. Para ello hace uso de redes neuronales, que consisten en un conjunto de unidades (neuronas) organizadas en capas y conectadas entre sí para transmitirse datos.

En el caso del procesamiento de una imagen que contiene el logo de Motul, ésta puede comenzar a ser procesada como un vector de valores de píxeles, y las características que son aprendidas en las primeras capas representan formas simples como líneas o colores. Las siguientes capas detectan patrones mediante la agrupación de bordes y aparecen formas tangibles como letras o círculos. Finalmente las últimas capas consiguen detectar el logo completo a través de la combinación de todas estas partes. En la Figura 29 encontramos un ejemplo de este procesamiento por capas.

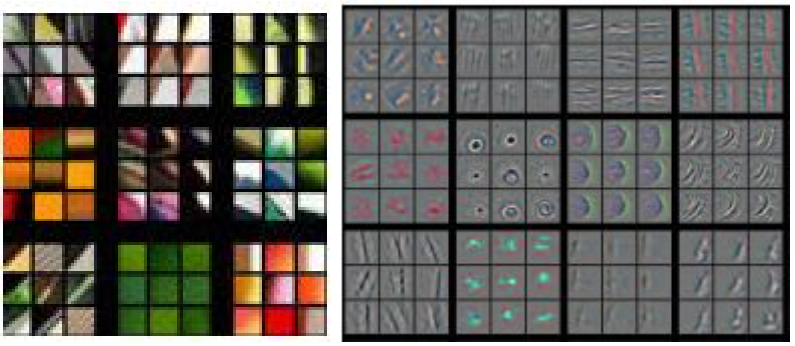


Figura 29. **Izquierda:** El algoritmo reconoce líneas y colores. **Derecha:** El algoritmo reconoce formas más complejas

El aspecto clave del *deep learning* es el hecho de que las características no son diseñadas por un humano, sino que son aprendidas a partir de las imágenes usando un procedimiento de aprendizaje de propósito general. En la Figura 30 se puede ver un diagrama que muestra la diferencia entre el enfoque tradicional y el basado en *deep learning*.

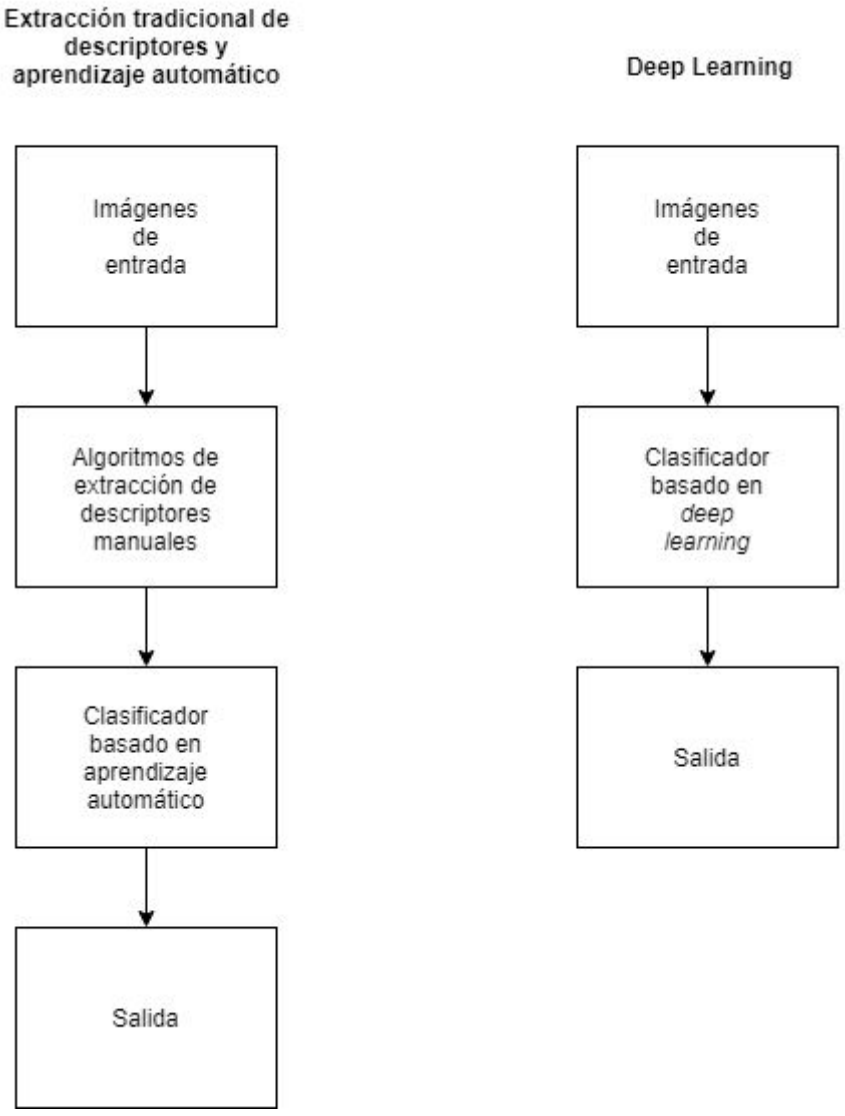


Figura 30. **Izquierda:** Proceso tradicional de obtener un conjunto de imágenes de entrada, aplicar algoritmos manuales de extracción de features (descriptores) y finalmente entrenar un clasificador de aprendizaje automático con los descriptores. **Derecha:** aproximación basada en *deep learning* en la que se apilan capas unas encima de otras y que automáticamente aprende descriptores más complejos, abstractos y discriminantes

A día de hoy el *deep learning* está presente en multitud de ámbitos a través de un gran número de aplicaciones como la medicina [16], la separación del audio de un interlocutor del ruido de fondo [17], la detección de peatones [18] o el reconocimiento de voz [19].

Para construir un modelo de detección de objetos basado en *deep learning* existen distintos algoritmos entre los que se encuentran: R-CNN [20], SSD [21] o YOLO [22]. En líneas generales en la arquitectura R-CNN prima la precisión a la hora de generar las detecciones a costa de un extenso tiempo de procesamiento. YOLO (*You Only Look Once*) presenta tiempos muy bajos de procesamiento permitiendo incluso el procesamiento en tiempo real aunque es algo menos precisa que R-CNN. Por último, la arquitectura SSD consigue un balance entre ambos. En nuestro caso la arquitectura que se va a usar es la de YOLO dado que su última versión YOLO v3 [23] ha conseguido un aumento notable en la precisión y la velocidad de procesado frente a las demás. Además se tiene experiencia previa con ella al haberla usado durante las prácticas en la empresa Pixelabs.

YOLO

La red YOLO aprende al mismo tiempo a localizar las coordenadas de los rectángulos delimitadores y a generar la probabilidad de que un objeto pertenezca a una clase. Esto se conoce como algoritmo de detección en una fase. Por lo general estos detectores tienden a ser menos precisos que los detectores en dos fases (que aprenden por separado a localizar y a generar la probabilidad), pero son mucho más rápidos procesando. Para poder trabajar con esta alta velocidad de procesamiento, sus creadores pensaron en aplicar la red neuronal a toda la imagen en lugar de a múltiples partes de esta, técnica que le dio nombre a la arquitectura (*You Only Look Once*, solo miras una vez). El sistema aplica una rejilla sobre la imagen dividiéndola en celdas, y cada región es la encargada de determinar si hay un objeto o no en su zona. Para cada celda se generan unas localizaciones posibles y un valor que se denomina confianza, y que determina la probabilidad que la red estima de que esa celda contenga el objeto de la clase que indica. Por tanto una detección está determinada por la posición de la predicción, su tamaño y su confianza. YOLO cuenta con varias versiones desde su publicación en 2015, como son la YOLO9000 [24] o la YOLO v3. Como hemos comentado en el apartado anterior, dado que presenta mejoras respecto a sus predecesoras, se va a usar la arquitectura YOLO v3.

4.2 Diseño

El proceso para entrenar un modelo de detección de objetos basado en la técnica YOLO consta de la obtención de un *dataset*, su anotación, la partición del *dataset* en los distintos conjuntos, la implementación del algoritmo basado en YOLO, el uso del conjunto de entrenamiento para crear el modelo y el uso del conjunto de validación para elegir los mejores parámetros del modelo. Como se puede ver, la mayoría de los pasos son iguales que en la implementación de la técnica HOG usando la librería dlib. Los únicos aspectos diferentes son el algoritmo de la red YOLO, que en este caso se hará mediante el *framework darknet* [25], y la transformación de las anotaciones ya que habrá que adecuarlas al formato que admite *darknet*. En la Figura 31 se muestra un esquema del proceso.

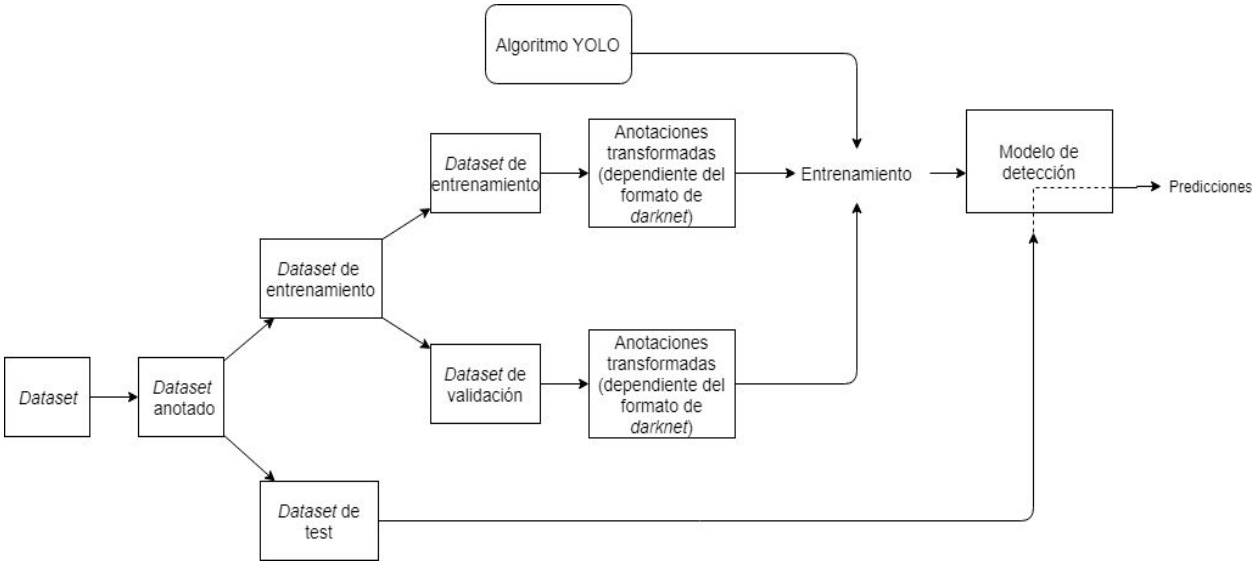


Figura 31. Proceso de creación de un modelo de detección de objetos basado en YOLO

Como se ha dicho en el rediseño de la técnica basada en HOG y SVM, el *dataset*, las anotaciones y la partición del *dataset* son independientes de la técnica que usemos y del algoritmo que implementemos, así que siguen siendo los mismos que al usar la librería dlib.

Framework

El mismo autor de la publicación de la red YOLO proporciona un *framework* llamado *darknet* a través del cual se simplifica mucho el uso de la red YOLO. En concreto se va a usar la implementación de este *framework* de AlexeyAB [26] que está más optimizada que la versión original y es más rápida. Este *framework* permite ejecutar las distintas versiones de la red y crear modelos de detección de objetos.

Transformación

Al igual que en la primera iteración pasó con dlib, el uso del *framework darknet* obliga a tener que procesar las anotaciones originales para que sean compatibles con el formato que admite este *framework*. En este caso, y como se ve en la Figura 32, el formato que usa *darknet* es el de un archivo con extensión `.txt` para cada imagen. Cada archivo contiene los objetos que se han identificado para esa imagen. Un objeto está compuesto por el identificador de su clase (0 en este caso al haber solo una clase), los valores x e y del centro del rectángulo delimitador y relativos a la altura y anchura de la imagen (por lo tanto valores entre 0 y 1), y la anchura y altura del rectángulo, también relativas al tamaño de la imagen. Nuevamente se ha elaborado un *script* que convierte las anotaciones originales al formato que acepta el *framework*.

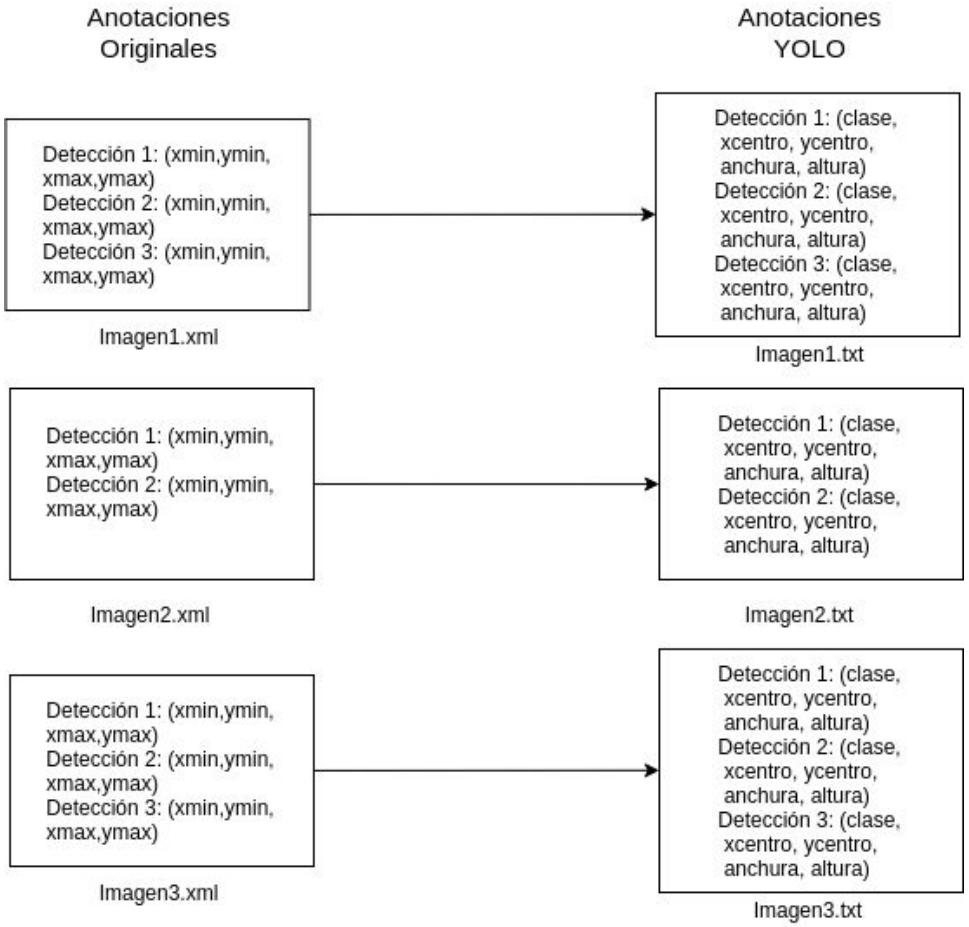


Figura 32. Esquema de transformación de anotaciones

4.3 Entrenamiento

Una vez se tiene el *dataset* adecuado al formato que admite el *framework darknet* se puede empezar a entrenar.

Requisitos

Los requisitos necesarios para la instalación y compilación de *darknet* son:

- Sistema operativo Ubuntu 16.04 o posterior. En nuestro caso se usará Ubuntu 16.04 LTS.
- Python. En nuestro caso se usará Python 3.6.
- OpenCV >= 2.4 (librería libre de visión por computador). En nuestro caso se usará OpenCV 3.4.0.

Con estos tres puntos ya se cumple lo necesario para que funcione el *framework darknet*. Sin embargo existe la posibilidad de usar una tarjeta gráfica (GPU) para paralelizar los procesos de entrenamiento y procesado. Tradicionalmente la capacidad de cómputo venía asociada al número de núcleos del procesador (CPU) que se tenía, los cuales están optimizados para el procesamiento en serie. Pero durante los últimos años la tendencia ha sido la de introducir las tarjetas gráficas como herramienta de procesamiento debido a los miles de núcleos que incorporan y a que están preparados para trabajar en paralelo muy eficientemente. Además las GPU trabajan muy bien en el procesamiento de imágenes, lo que es una de las tareas centrales del proyecto. Para poder aprovechar el uso de una GPU, se va a trabajar en el mismo ordenador con 16GB de RAM que se ha usado para el entrenamiento usando la técnica HOG, y que cuenta con una tarjeta gráfica Nvidia GTX 1080 Ti (la cual no era necesaria para el entrenamiento usando la técnica HOG). Para poder usarla se tienen que instalar:

- *Drivers* de Nvidia
- Cuda >= 10.0 (conjunto de herramientas para procesamiento de datos de Nvidia).
- CuDNN >= 7.0 (librería para redes neuronales profundas de Nvidia) para Cuda 10.0, distinto en caso de instalar una versión superior de Cuda. En nuestro caso se usará CuDNN 7.1.3.
- CMake >= 3.8 (librería para controlar el compilado). En nuestro caso se usará CMake 3.8.
- GPU con CC (*compute capability*) >= 3.0. En nuestro caso la tarjeta gráfica de la que se dispone tiene la versión 6.1.
- GCC (compilador de GNU).

Parámetros y configuración

El usuario AlexeyAB en su implementación de *darknet* da unos consejos a la hora de modificar los parámetros para realizar el entrenamiento. Para ello hay que crear los archivos `class.cfg`, `class.data`, `class.names`, `train.txt` y `validate.txt`. En primer lugar se crea el archivo `class.cfg` con el mismo contenido que el archivo `yolov3.cfg` que se encuentra en la subcarpeta `cfg` del directorio `darknet`. Este archivo contiene los parámetros de configuración que se usan para el entrenamiento de la red YOLO como se ve en la Figura 33, así como la arquitectura de las distintas capas que se usarán durante el entrenamiento. No

se va a profundizar en las especificaciones de las distintas capas ya que se sale del alcance del proyecto.

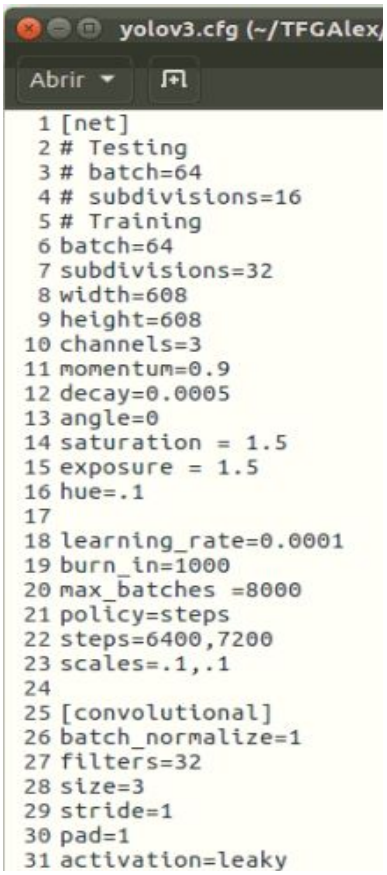


Figura 33. Archivo yolov3.cfg

Los parámetros de este archivo que el autor recomienda modificar con el fin de optimizar el entrenamiento son `batch`, `subdivisions`, `max_batches`, `steps`, `classes`, `filters` y `learning-rate`:

- El parámetro `batch` determina la cantidad de imágenes que el modelo cargará para entrenar en cada iteración. Recomienda asignarle el valor 64.
- El parámetro `subdivisions` establece la fracción de cada iteración que será mandada a la GPU para su procesamiento, evitando desbordamientos de memoria. Recomienda asignarle el valor 8. En este caso se opta por asignar el valor 32 para prevenir desbordamientos. De esta forma en cada `subdivision` se mandarán a la GPU 64/32=2 imágenes.
- El parámetro `max_batches` determina el máximo de iteraciones que se completarán antes de dar fin al entrenamiento. Recomienda asignarle el valor nº de clases * 2000. Adicionalmente recomienda no entrenar el modelo con menos de 4000 `batches`. En nuestro caso le asignamos el valor 8000.
- El parámetro `steps` determina tras qué iteraciones se modificará el valor del parámetro `learning_rate` multiplicándolo por el valor del parámetro `scales`. Recomienda asignarle el 80% y 90% del valor del parámetro `max_batches`. En nuestro caso, le asignamos el valor 8000 * 0.8=6400 y 8000 * 0.9=7200. Por defecto el parámetro `scales` tiene el valor 0.1 y 0.1 (a las 6400 iteraciones el `learning_rate` se multiplicará por 0.1, y a las 7200 iteraciones por 0.1 otra vez).

- El parámetro `classes` de cada capa YOLO. Recomienda asignarle el número de clases distintas que se van a detectar. En nuestro caso asignamos el valor 1.
- El parámetro `filters` de las tres capas anteriores de las capas YOLO. Recomienda asignarle el valor resultante de (`classes` + 5) * 3. En nuestro caso asignamos el valor (1+5) * 3 = 18.
- El parámetro `learning_rate` define la magnitud con la que se van a actualizar los pesos de la red tras cada iteración. Estos pesos son los valores que modifican los datos que se transmiten las neuronas entre sí. Inicialmente lo dejamos en 0.0001.

Una vez entrenado el primer modelo y constatado que todo funciona bien, se aplicará de nuevo la técnica *GridSearch* para encontrar el que mejores resultados genere. En este caso se va a probar con `learning_rate` igual a 0.0001, 0.001, 0.01 y 0.1. El parámetro `max_batches` se va a mantener con el valor 8000 ya que los resultados se pueden ir consultando cada 1000 iteraciones, y de esta forma se pueden recoger los resultados con el parámetro variando entre los valores 1000, 2000, … 7000 y 8000.

El segundo archivo que hay que configurar es el `class.names`, que contiene los nombres de las clases de los objetos que se van a detectar. En nuestro caso creamos el archivo con una sola línea que contiene la palabra “motul”.

El siguiente archivo es el `class.data` y contiene las líneas que se ven en la Figura 34.

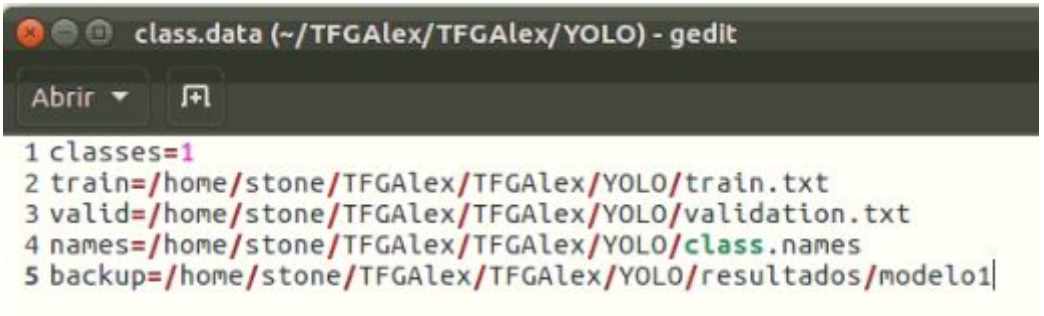


Figura 34. Archivo class.data

El valor de `classes` es el del número de clases con las que se va a entrenar. Los valores de `train` y `valid` son las rutas a dos archivos que se crearán más adelante. El valor de `names` es la ruta al archivo `class.names`. Y el valor de `backup` es la ruta al directorio donde se irán guardando los pesos. Cada 100 iteraciones se actualizará un fichero llamado `yolo-obj_last.weights`, por lo que si en cualquier momento se para el entrenamiento contendrá los pesos de la última iteración múltiplo de 100. Además cada 1000 iteraciones se generará un fichero con nombre `yolo-obj_xxxx.weights`. Como se ha explicado anteriormente estos pesos son los valores que modifican la información que se va transmitiendo entre neuronas.

El siguiente paso consiste en ubicar las imágenes de entrenamiento en un directorio y proceder a anotarlas. Dado que ya se ha realizado la transformación de las anotaciones originales a las requeridas por YOLO, basta con juntar las imágenes y los ficheros con las anotaciones en el mismo directorio. Debido a que se va a usar un conjunto de validación ahora se tiene que crear además del fichero `train.txt` el fichero `validate.txt`. Estos ficheros contienen las rutas de las imágenes que se van a usar para el entrenamiento y para la validación respectivamente con cada ruta relativa al ejecutable `darknet` en una línea. Para ello se elabora un script que genere el archivo. Para poder comenzar el entrenamiento hay que descargar unos pesos

preentrenados que son genéricos para iniciar el entrenamiento de cualquier modelo usando la red YOLO. Estos pesos vienen definidos en el archivo `darknet53.conv.74`. Tras completar estos pasos el entrenamiento está listo para ser lanzado.

Inicio del entrenamiento

Para iniciar el entrenamiento se ejecuta el comando `./darknet detector train ruta/al/class.data ruta/al/class.cfg darknet53.conv.74 -map`. La terminal comienza entonces a mostrar información de cómo va aprendiendo el modelo en cada iteración. Como se ve en la Figura 35 la terminal muestra la pérdida, la pérdida media de las últimas iteraciones (no se especifica el número de iteraciones), el *learning rate* en esa iteración, los segundos que ha tardado y el número de imágenes que lleva procesadas. Al usar el parámetro `-map` después de la iteración número 1000 y cada 125 iteraciones desde ese momento la red calcula la métrica mAP.

```
(next mAP calculation at 3625 iterations)
Last accuracy mAP@0.5 = 89.28 %
3582: 1.112975, 1.000911 avg loss, 0.000010 rate, 8.019018 seconds, 229248 images
```

Figura 35. Salida de terminal durante el entrenamiento.

Salidas

Las salidas que produce el *framework darknet* y que indican el comportamiento del modelo son la pérdida (*loss*), las ya conocidas *precision*, *recall* y *F1 score*, y la métrica mAP (*mean Average Precision*). La pérdida viene definida por la función de pérdida, que es una función usada por las redes neuronales para evaluar una combinación de pesos. Esta función utiliza las predicciones hechas en cada iteración de entrenamiento y las compara con las que se le han dado al modelo para evaluar los pesos en ese momento. De esta forma calcula un valor que es la pérdida, la cual es menor cuanto mejor ha aprendido el modelo hasta ese momento, y por tanto el objetivo de la red es el de minimizar este valor. El parámetro del que va a depender que la red aprenda bien y se minimice la pérdida es el *learning rate*, que como se ha dicho antes define cómo se van ajustando los pesos tras cada iteración. Es importante encontrar un buen valor para este parámetro, ya que si es demasiado alto los cambios en los pesos serán demasiado bruscos y puede que se salte el punto de pérdida mínima. Si por el contrario el parámetro tiene un valor demasiado pequeño, los cambios en los pesos también lo serán, y puede llevar un tiempo inmenso alcanzar el mínimo (el modelo acabará de entrenar cuando llegue al número máximo de iteraciones definido sin haber llegado todavía al punto óptimo). Por tanto los parámetros con los que se busca optimizar el modelo son `learning_rate` y `max_batches`, que son sobre los que se aplicará la técnica *GridSearch*.

Por otro lado la métrica mAP es una de las más usadas en el ámbito de la detección de objetos. Esta métrica se obtiene a partir de la *Average Precision*, que se calcula como media de las precisiones en todas las imágenes procesadas con respecto a una clase, como se ve en la Figura 36.

$$AveragePrecision_C = \frac{\sum_{i=1}^{N_C} Precision_i}{N_C}$$

Figura 36. Fórmula de la métrica *Average Precision*

Calculando la media de la *Average Precision* para todas las clases del *dataset* se obtiene la métrica mAP o *mean Average Precision*, como se ve en la Figura 37.

$$mAP = \frac{\sum_{C \in C_{tot}} AveragePrecision_C}{N_C}$$

Figura 37. Fórmula de la métrica mAP

Durante el entrenamiento se va generando una imagen en la que se ve cómo han ido evolucionando el valor de la pérdida (puntos discontinuos) y el de la métrica mAP (línea continua) a lo largo de las iteraciones como se puede ver en la Figura 38.

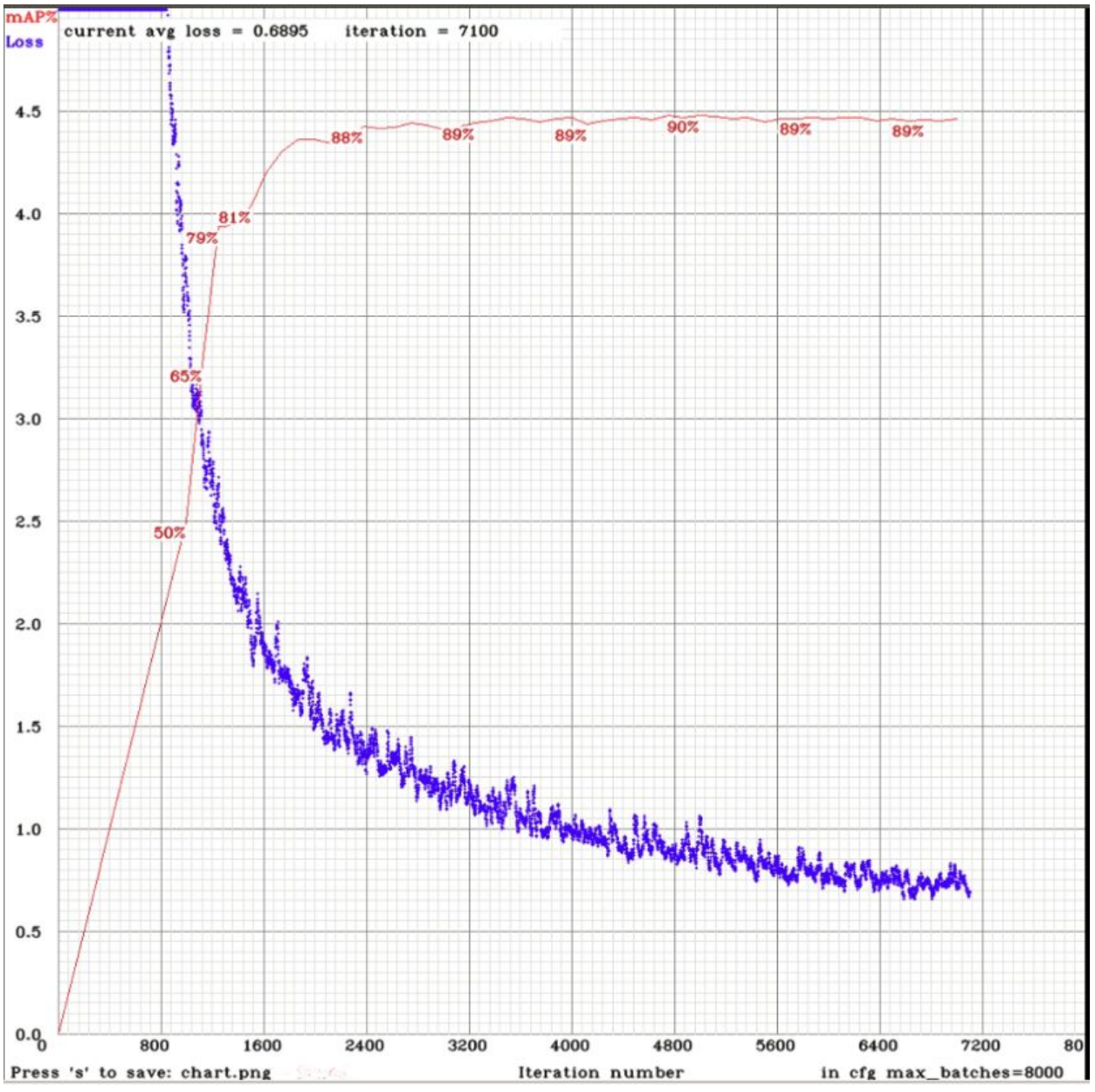


Figura 38. Gráfico generado por el *framework darknet*

Resultados

Tras entrenar todos los modelos se han obtenido los resultados que se ven en la Tabla 6. A la vista de los valores registrados para las métricas se ve que los resultados son bastante buenos en casi todos los modelos. Como mejor modelo se elige el entrenado con *learning rate* de 0.01 y 8000 *batches* ya que consigue la mejor métrica *F1 score* y la mejor métrica mAP. El entrenamiento de todos los modelos ha durado 3 días, 20 horas y 24 minutos. El tiempo medio de entrenamiento para cada modelo ha sido de 18 horas y 28 minutos.

Parámetros		Resultados		
Learning Rate	Batches	Pérdida	F1 Score	mAP
0,1	1000	NaN	0	0%
0,1	2000	NaN	0	0%
0,1	3000	NaN	0	0%
0,1	4000	NaN	0	0%
0,1	5000	NaN	0	0%
0,1	6000	NaN	0	0%
0,1	7000	NaN	0	0%
0,1	8000	NaN	0	0%
0,01	1000	0,7521	0,92	89%
0,01	2000	0,3553	0,92	89,3%
0,01	3000	0,3212	0,93	90,1%
0,01	4000	0,2531	0,93	90%
0,01	5000	0,2481	0,94	90,4%
0,01	6000	0,2067	0,94	90,7%
0,01	7000	0,1617	0,95	90,7%
0,01	8000	0,1512	0,97	91,2%
0,001	1000	0,8196	0,89	87,8%
0,001	2000	0,4140	0,90	89,6%
0,001	3000	0,3174	0,91	90,1%
0,001	4000	0,2638	0,91	90,1%
0,001	5000	0,2308	0,92	90,3%
0,001	6000	0,2195	0,91	90,2%
0,001	7000	0,1734	0,93	90,5%
0,001	8000	0,1701	0,93	90,5%
0,0001	1000	1,1021	0,88	88,2%
0,0001	2000	0,7269	0,89	89,1%
0,0001	3000	0,5108	0,88	89,4%
0,0001	4000	0,4767	0,90	89,6%
0,0001	5000	0,4018	0,90	89,5%
0,0001	6000	0,3512	0,92	89,7%
0,0001	7000	0,3304	0,93	89,9%
0,0001	8000	0,3069	0,93	90,2%
0,00001	1000	4,513	0,45	60,4%
0,00001	2000	2,7451	0,81	85,3%
0,00001	3000	1,2677	0,87	87,5%
0,00001	4000	1,0021	0,89	87,8%
0,00001	5000	0,9369	0,89	88,4%
0,00001	6000	0,7769	0,90	88,3%
0,00001	7000	0,7010	0,91	88,8%
0,00001	8000	0,6587	0,91	89,3%

Tabla 6. Resultados de los modelos entrenados usando la técnica YOLO

5. Comparativa

En esta iteración se elegirán las métricas con las que se compararán los modelos y se realizará la comparativa entre los modelos generados.

5.1 Métricas

Las métricas más importantes en el ámbito de la detección de objetos son las métricas mAP y *F1 score* (en cuanto a tasa de acierto) y el número de FPS (en cuanto a velocidad de procesado). Además se añadirán a la comparativa otros aspectos como requerimientos para implementar las técnicas y tiempo de entrenamiento de los modelos.

Precisión

Para medir la precisión se van a usar la métrica *F1 score* y la métrica mAP. Esta segunda es la métrica por excelencia usada para medir el rendimiento de un modelo de detección de objetos, y para su cálculo se va a utilizar la librería "mAP" [27] disponible en GitHub. Para poder utilizar esta librería en primer lugar se deben formatear las detecciones del conjunto de test como se puede ver en la Figura 39 y situarlas en la carpeta `input/ground-truth`.

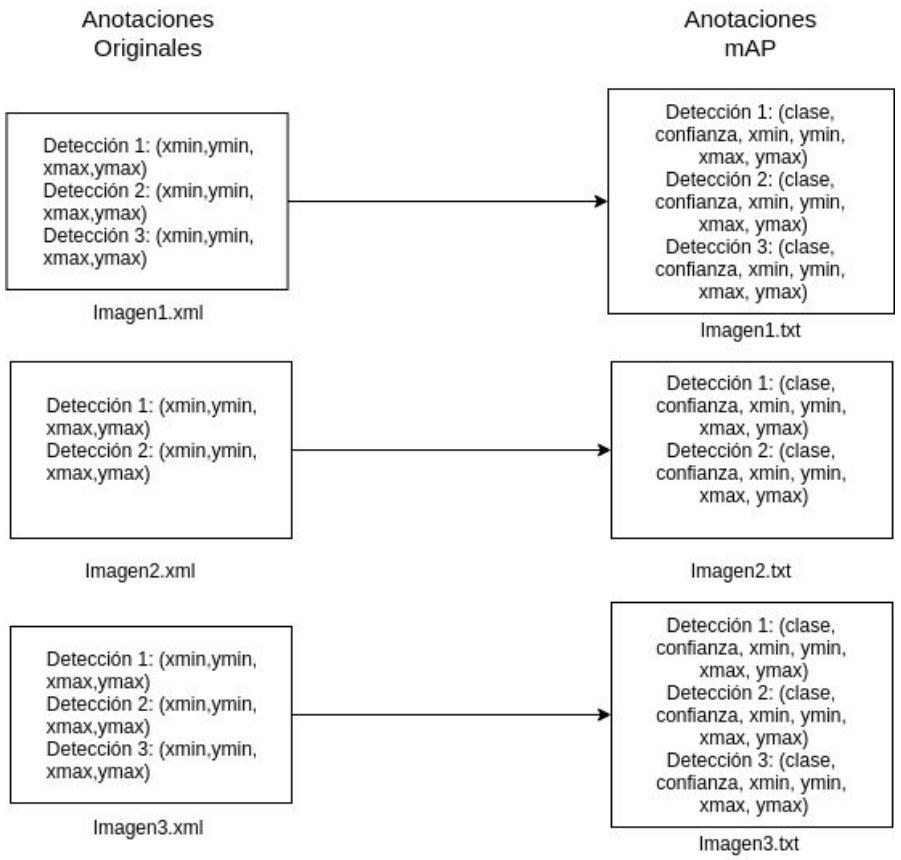


Figura 39. Esquema de transformación de anotaciones

Una vez se tienen estas anotaciones hay que generar las predicciones con el modelo de detección, que se ubicarán en la carpeta `input/detection-results`. En el caso del modelo basado en la técnica HOG (en el que el código para el entrenamiento y el procesamiento es propio) se ha implementado una salida acorde con el formato de la librería. En el caso del

modelo basado en la técnica YOLO se ha elaborado un nuevo *script* que hace la transformación entre el formato de salida de *darknet* y el que usa la librería de cálculo de la métrica mAP.

Velocidad de procesado

Para medir la velocidad de procesado se va a utilizar la métrica FPS, que indica el número de imágenes que el modelo es capaz de procesar por segundo y toma valores entre 0 e infinito. Para cuantificar este valor se va a calcular el tiempo que el modelo tarda en procesar el total de imágenes del conjunto de test, y se dividirá por el número de imágenes de dicho conjunto.

Otros aspectos

Adicionalmente se van a comparar los requerimientos de sistema que se necesitan para crear los modelos. También se van a añadir a la comparativa los tiempos necesarios para completar el entrenamiento de cada modelo.

5.2 Evaluación de los modelos

Una vez descritas las métricas, se van a evaluar los modelos.

Precisión

En primer lugar se calcula la métrica mAP que obtiene cada modelo sobre el conjunto de test. En el caso de la técnica HOG implementada a mano, se obtiene una mAP de 68.38% y en el de YOLO una mAP de 85.72%. Si comparamos la métrica *F1 score* obtenida por ambos modelos, encontramos un valor de 0.63 para el modelo basado en HOG, frente al valor de 0.91 para el modelo basado en YOLO. Para visualizar un poco mejor estos resultados se pueden ver las Figuras 40, 41 y 42, que contienen en la parte izquierda los resultados obtenidos mediante la técnica HOG y en la derecha los resultados obtenidos mediante la técnica YOLO.

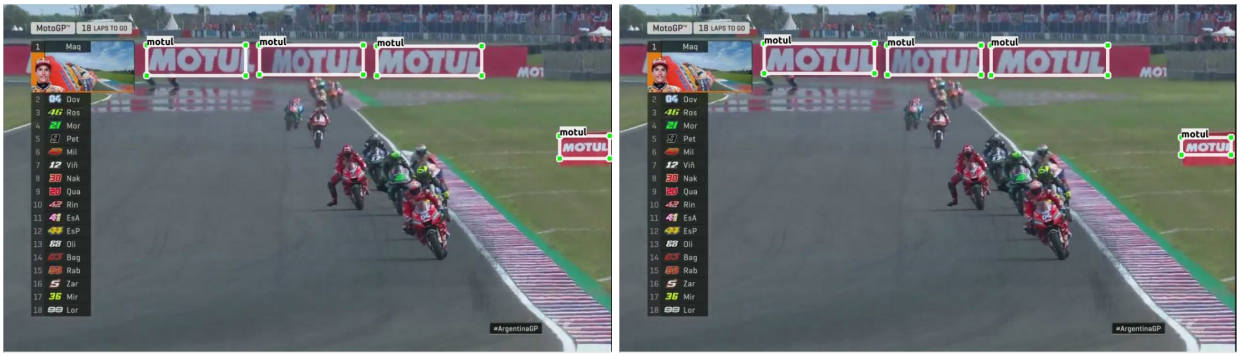


Figura 40. **Izquierda:** Resultado obtenido mediante la técnica HOG. **Derecha:** Resultado obtenido mediante la técnica YOLO

Figura 41. **Izquierda:** Resultado obtenido mediante la técnica HOG. **Derecha:** Resultado obtenido mediante la técnica YOLO

Figura 42. **Izquierda:** Resultado obtenido mediante la técnica HOG. **Derecha:** Resultado obtenido mediante la técnica YOLO

Los resultados obtenidos con el modelo basado en HOG son bastante buenos, sin embargo en algunos casos la detección no termina de ajustarse del todo al logo de Motul. En la parte izquierda de la Figura 40 se puede ver que el modelo ha respondido bien a los distintos tamaños del logo gracias al uso de la ventana deslizante y la pirámide de imágenes implementadas. El rendimiento se ve afectado por difuminaciones en el logo como muestra la parte izquierda de la Figura 41, donde solo es capaz de reconocer uno de los tres logos que hay. Una posible solución a esto habría sido añadir un filtro de desenfoque a las imágenes de entrada para aumentar el conjunto de entrenamiento. Sin embargo es incluso sorprendente cómo en la parte izquierda de la Figura 42 es capaz de reconocer el logo tan solo por las letras “MO” del mismo. Por contra, esto último es una de las razones por las que la métrica mAP se ve penalizada, ya que tanto en el conjunto de entrada como en el conjunto de test solo se etiquetaron aquellos logos que se veían al completo, por tanto al modelo se le pedía que detectara logos completos de Motul.

Los resultados obtenidos con el modelo basado en YOLO en este caso son muy buenos. En la parte derecha de la Figura 40 se puede ver cómo el modelo detecta todos los logos con gran precisión. En la parte derecha de la Figura 41 se puede ver cómo este modelo responde mejor ante las difuminaciones logrando detectar dos de los tres logos de Motul. Al igual que el modelo HOG, es capaz de reconocer incluso fragmentos del logo original, tan solo con las letras “MO” o tan solo con las letras “OTUL” como se ve en la parte derecha de la Figura 42, lo que nuevamente repercute negativamente en la métrica mAP que obtiene.

Por tanto la conclusión a la que se puede llegar es que el modelo basado en la técnica YOLO es más preciso que el basado en la técnica HOG (un 17% en este caso). Esta diferencia es significativa y es algo que se esperaba dado que la técnica YOLO es bastante más sofisticada que las técnicas HOG y SVM. Además el modelo basado en YOLO responde mucho mejor a las pocas variaciones que hay en el logo, como en este caso es el desenfoque.

Velocidad de procesado

El tiempo que ha tardado el modelo basado en HOG en procesar las 400 imágenes ha sido de 1 minutos y 53 segundos. Por tanto, su velocidad de procesado es de 3.5 FPS.

Por otro lado, el tiempo que que ha tardado el modelo basado en YOLO en procesar las 400 imágenes ha sido de 11 segundos. Por tanto su velocidad de procesado es de 36.4 FPS.

Las consecuencias de esto son que el modelo de HOG no puede ser empleado en un sistema que trabaje en tiempo real (40 FPS) dada su reducida velocidad, al contrario que el modelo basado en YOLO que consigue casi esa velocidad de procesamiento.

Otros aspectos

Si bien es cierto que en las anteriores métricas el modelo basado en YOLO supera al basado en HOG, esto es en parte por el uso de la tarjeta gráfica que le permite una potencia de cálculo mucho mayor. Esto es algo a tener en cuenta ya que si por ejemplo se quisieran implementar ambos modelos en un sistema sencillo, como podría ser una *Raspberry Pi*, la velocidad de procesado de YOLO se vería bastante reducida mientras que la de HOG no variaría casi. Es por ello que hay que notar que, por un lado el modelo basado en HOG tan solo ha necesitado de una máquina con Ubuntu y una librería para entrenar el modelo, mientras que el modelo basado en YOLO ha trabajado adicionalmente con una tarjeta gráfica y con una serie de librerías que le han permitido usarla para el entrenamiento del modelo y el posterior procesado de imágenes. Además estas librerías y *drivers* (CUDA, CuDNN, *drivers* de Nvidia) no son para nada triviales de instalar, y por ello la puesta en marcha del entrenamiento de YOLO requiere de un trabajo previo para que todo funcione correctamente.

En cuanto al tiempo de entrenamiento mientras que HOG ha necesitado 1 hora y 12 minutos YOLO ha necesitado más de 15 veces más tiempo para completar su entrenamiento, durando este 18 horas y 28 minutos.

Conclusiones de la comparativa

Tras haber analizado ambos modelos se ha llegado a varias conclusiones. Analizando la precisión que han alcanzado ambos modelos se puede decir que las dos técnicas cumplen bastante bien con el objetivo de la detección del logo de Motul. El hecho de que el logo no sufra grandes alteraciones tanto en iluminación, como en forma, o posición favorece mucho el trabajo del modelo basado en HOG, haciendo que su rendimiento no diste demasiado del modelo basado en YOLO que sí que está mejor preparado para estas alteraciones. Sin embargo es en la velocidad de procesado donde se observan las diferencias más cruciales. El procesado de una carrera completa de 1 hora mediante el modelo HOG duraría 15 horas, mientras que mediante el modelo YOLO duraría tan solo 1 hora y podría incluso realizarse en tiempo real. La pega de este mayor rendimiento se encuentra tanto en el tiempo de entrenamiento del modelo (que como se ha visto es alrededor de 15 veces mayor, 1 hora y 12 minutos para HOG frente a 18 horas y 28 minutos para YOLO) y en los requerimientos extras que necesita el modelo

basado en YOLO, como son el disponer de una GPU y configurar las distintas librerías para su funcionamiento.

Por tanto, para un sistema de detección de logos fácil de implementar y que no precise de gran rapidez a la hora de procesar las imágenes se optaría por la técnica HOG. Por el contrario, para una precisión algo mayor y poder trabajar incluso con vídeo en tiempo real se optaría por la técnica YOLO, haciendo falta un ordenador con una GPU configurada.

6. Seguimiento y control

La Figura 43 muestra el diagrama de Gantt con los tiempos previstos para la realización del proyecto y el tiempo real empleado para cada tarea. El proyecto se ha completado aun habiendo surgido imprevistos que han supuesto alteraciones en el plan propuesto:

- Durante las primeras semanas la mayoría del tiempo no pudo invertirse en el proyecto dada la carga de trabajo. Además la planificación llevó 15 horas más de las previstas. Por ello la primera iteración comenzó el 8 de abril en vez del 4 de marzo como estaba planificado.
- Durante la primera iteración aparecieron problemas al usar la librería dlib provocando un aumento de las horas previstas de 10 a 40. Finalmente se vio que la librería no servía en este caso y hubo que implementar la técnica basada en HOG a mano, lo cual supuso unas 60 horas adicionales. Esta nueva implementación se solapó con la segunda iteración dado que se sabía de antemano que los entrenamientos podían ser largos y de esta forma se distribuyó mejor el tiempo.
- La segunda iteración sin embargo requirió de 50 horas menos dado que no hubo grandes problemas con la instalación tanto de las librerías como del *framework darknet*.
- Se han dedicado 345 de las 285 horas previstas (a falta de la preparación de la defensa del TFG).

ID	Tarea	Inicio	Fin	Duración
1	Gestión	11/02/19	21/06/19	80
		11/02/19	21/06/19	105
1.1	Planificación	11/02/19	04/02/19	35
		11/02/19	04/02/19	50
1.2	Seguimiento	11/02/19	21/06/19	10
		11/02/19	21/06/19	10
1.3	Memoria	11/02/19	21/06/19	25
		11/02/19	21/06/19	35
1.4	Alcance	11/02/19	04/02/19	5
		11/02/19	04/02/19	5
1.5	Requisitos	11/02/19	04/02/19	5
		11/02/19	04/02/19	5
2	Iteración 1 (HOG)	04/03/19	19/04/19	90
		08/04/19	07/06/19	175
2.1	Análisis	04/03/19	15/03/19	25
		08/04/19	03/05/19	50
2.2	Diseño	15/03/19	12/04/19	55
		06/05/19	10/05/19	25
2.3	Entrenamiento	12/04/19	19/04/19	10
		13/05/19	24/05/19	40
2.4	Rediseño	27/05/19	29/05/19	40
2.5	Entrenamiento	29/05/19	31/05/19	20
3	Iteración 2 (YOLO)	22/04/19	07/06/19	90
		19/04/19	07/06/19	40
3.1	Análisis	19/04/19	03/05/19	25
		20/05/19	22/05/19	10
3.2	Diseño	03/05/19	31/05/19	55
		22/05/19	29/05/19	20
3.3	Entrenamiento	31/05/19	07/06/19	10
		29/05/19	03/06/19	10
4	Iteración 3 (Evaluación)	10/06/19	21/06/19	25
		03/06/19	14/06/19	25
4.1	Métricas para la evaluación	03/06/19	05/06/19	15
		07/06/19	14/06/19	10
4.2	Comparativa	14/06/19	21/06/19	10
		05/06/19	14/06/19	15
6	Defensa del proyecto	24/06/19	11/07/19	16
6.1	Presentación	21/06/19	11/07/19	15
6.2	Vídeos	21/06/19	11/07/19	1

Figura 43. Diagrama de Gantt corregido., en negro dedicación prevista, en naranja dedicación real.

7. Conclusiones

En este apartado se van a presentar las conclusiones obtenidas tras la realización del proyecto.

La ventaja de las técnicas *deep* en la tarea de la detección de objetos

El hecho de que la gran mayoría de sistemas de detección de objetos esté basado en aprendizaje profundo no es casualidad. En este trabajo hemos podido analizar una técnica tradicional que ha obtenido un rendimiento bastante bueno, pero no es para nada comparable al obtenido con la red YOLO. Ya no solo en cuanto a la tasa de acierto, la cual en muchas aplicaciones es crítica (como por ejemplo la detección de cáncer a través de imágenes médicas), sino también en una velocidad de procesado que está a años luz de las alternativas tradicionales, permitiendo incluso el procesamiento en tiempo real abriendo un gran abanico de posibilidades.

También hay que tener en cuenta que el soporte que la comunidad ofrece para las nuevas técnicas que van apareciendo es inmensamente mayor que el que hay para las tradicionales. La gran cantidad de aplicaciones basadas en aprendizaje profundo que están surgiendo hace que la información que se puede encontrar sobre estas técnicas en la red sea muy grande, y por tanto en la práctica resultan incluso más fáciles de utilizar que las técnicas tradicionales pese a que su complejidad es mucho mayor.

La importancia de las GPUs

Parte del mérito de que las técnicas *deep* superen a las tradicionales es el uso que hacen de la tarjeta gráfica. La capacidad de cálculo que nos dan debido a sus miles de núcleos trabajando en paralelo hace que el cálculo mediante CPU, restringido a unos pocos núcleos, no pueda competir con ellas. Sin duda éste ha sido uno de los grandes avances que han surgido durante los últimos años y que ha permitido que la inteligencia artificial dé un gran paso hacia delante.

El papel de los *datasets*

Pese a que los *datasets* parecen la parte más arcaica y manual de un modelo de detección de objetos, son los cimientos del buen funcionamiento de los modelos. El tiempo extra dedicado a conseguir un *dataset* bueno y consistente será tiempo que ahorraremos después en modelos que no funcionen.

Posiblemente el trabajo más costoso en la tarea de la detección de objetos sea la anotación del *dataset*. He tenido la suerte de poder utilizar un *dataset* ya anotado para la elaboración del trabajo, pero durante las prácticas que realicé en la empresa Pixelabs tuve que formar parte del equipo que realizaba esta anotación. Para que nos hagamos una idea, este *dataset* (con un total de 2500 imágenes) necesitó de 20 horas de trabajo de una persona para ser anotado, utilizando una herramienta modificada por la propia empresa para facilitar este proceso.

Además del coste que supone esta anotación, es muy importante tener cuidado a la hora de hacer las diferentes particiones del *dataset*. La división tiene que ser representativa del conjunto ya que si no, como en el caso de los modelos que se crearon usando la librería dlib, puede parecer que los resultados son buenos cuando en realidad no lo son. En dicho caso, el conjunto de validación no contenía suficientes logos de un tamaño pequeño, y daba la impresión de que los modelos funcionaban bien cuando en realidad no eran capaces de

detectarlos. Por tanto hay que dedicar tiempo a gestionar la división del *dataset* y a hacer que los tres conjuntos resultantes sean representativos del conjunto inicial.

Experiencia personal

La realización de este trabajo me ha supuesto una gran experiencia a nivel personal. He aprendido lo importante que es saber pivotar y buscar alternativas ante los problemas que puedan surgir, y no seguir trabajando en una vía muerta como era el caso de la librería dlib, ya que tras obtener los modelos con el comportamiento extraño la mejor opción fue la de pasar a implementar el código por mi cuenta. He podido experimentar las dificultades de implementar con éxito distintas técnicas de visión por computador. Además las primeras semanas del proyecto tuve que destinar la mayor parte de mi tiempo a otros trabajos, lo cual me hizo tener que reorganizar el tiempo que tenía planificado para el proyecto y realizar distintos ajustes. Mis conocimientos de Python han aumentado con creces y me he sentido muy cómodo trabajando en este lenguaje y consiguiendo entrenar con éxito modelos de detección. Aunque se han obtenido unos resultados interesantes, el análisis de otras técnicas tradicionales y *deep* podría ser una ampliación muy interesante para este proyecto.

Bibliografía

1. Boscacci R. What Even is Computer Vision? In: Towards Data Science [Internet]. 16 Dec 2018 [cited 4 Mar 2019]. Available: https://towardsdatascience.com/what-even-is-computer-vision-531e4f07d7d0

2. Luozm. Different Tasks in Computer Vision. In: Luozm Github [Internet]. 10 Sep 2017 [cited 28 Feb 2019]. Available: https://luozm.github.io/cv-tasks

3. Oladipupo T. Machine Learning Overview [Internet]. New Advances in Machine Learning. 2010. doi:10.5772/9374

4. Panchal PM, Panchal SR, Shah SK. A comparison of SIFT and SURF. International Journal of Innovative Research in Computer and Communication Engineering. pdfs.semanticscholar.org; 2013;1: 323–327.

5. Wilson PI, Fernandez J. Facial Feature Detection Using Haar Classifiers. J Comput Sci Coll. USA: Consortium for Computing Sciences in Colleges; 2006;21: 127–133.

6. Ahonen T, Hadid A, Pietikäinen M. Face description with local binary patterns: application to face recognition. IEEE Trans Pattern Anal Mach Intell. computer.org; 2006;28: 2037–2041.

7. Dalal N, Triggs B. Histograms of oriented gradients for human detection. international Conference on computer vision & Pattern Recognition (CVPR'05). IEEE Computer Society; 2005. pp. 886–893.

8. Liaw A, Wiener M, Others. Classification and regression by randomForest. R news. researchgate.net; 2002;2: 18–22.

9. Rosebrock A. k-NN classifier for image classification - PyImageSearch. In: PyImageSearch [Internet]. 8 Aug 2016 [cited 8 May 2019]. Available: https://www.pyimagesearch.com/2016/08/08/k-nn-classifier-for-image-classification/

10. Kumari R. SVM classification an approach on detecting abnormality in brain MRI images. Int J Eng Res Appl. pdfs.semanticscholar.org; 2013;3: 1686–1690.

11. Lawrence S, Giles CL, Tsoi AC, Back AD. Face recognition: a convolutional neural-network approach. IEEE Trans Neural Netw. cs.cmu.edu; 1997;8: 98–113.

12. Cao X, Wu C, Yan P, Li X. Linear SVM classification using boosting HOG features for vehicle detection in low-altitude airborne videos. 2011 18th IEEE International Conference on Image Processing. ieeexplore.ieee.org; 2011. pp. 2421–2424.

13. Rosebrock A. Deep Learning for Computer Vision with Python: Practitioner Bundle. 2017.

14. davisking. davisking/dlib. In: GitHub [Internet]. [cited 10 May 2019]. Available: https://github.com/davisking/dlib

15. LeCun Y, Bengio Y, Hinton G. Deep learning. Nature. 2015;521: 436–444.

16. Lee J-G, Jun S, Cho Y-W, Lee H, Kim GB, Seo JB, et al. Deep Learning in Medical Imaging: General Overview. Korean J Radiol. synapse.koreamed.org; 2017;18: 570–584.

17. Wang D, Chen J. Supervised Speech Separation Based on Deep Learning: An Overview. IEEE/ACM Transactions on Audio, Speech, and Language Processing. ieeexplore.ieee.org; 2018;26: 1702–1726.

18. Ouyang W, Wang X. Joint deep learning for pedestrian detection. Proceedings of the IEEE International Conference on Computer Vision. openaccess.thecvf.com; 2013. pp. 2056–2063.

19. Amodei D, Ananthanarayanan S, Anubhai R, Bai J, Battenberg E, Case C, et al. Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin. International Conference on Machine Learning. jmlr.org; 2016. pp. 173–182.

20. Girshick R, Donahue J, Darrell T, Malik J. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation [Internet]. 2014 IEEE Conference on Computer Vision and Pattern Recognition. 2014. doi:10.1109/cvpr.2014.81

21. Liu W, Anguelov D, Erhan D, Szegedy C, Reed S, Fu C-Y, et al. SSD: Single Shot MultiBox Detector [Internet]. Computer Vision – ECCV 2016. 2016. pp. 21–37. doi:10.1007/978-3-319-46448-0_2

22. Redmon J, Divvala S, Girshick R, Farhadi A. You Only Look Once: Unified, Real-Time Object Detection [Internet]. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 2016. doi:10.1109/cvpr.2016.91

23. Redmon J, Farhadi A. YOLOv3: An Incremental Improvement [Internet]. arXiv [cs.CV]. 2018. Available: http://arxiv.org/abs/1804.02767

24. Redmon J, Farhadi A. YOLO9000: Better, Faster, Stronger [Internet]. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 2017. doi:10.1109/cvpr.2017.690

25. Redmon J. YOLO: Real-Time Object Detection [Internet]. [cited 24 May 2019]. Available: https://pjreddie.com/darknet/yolo/

26. AlexeyAB. AlexeyAB/darknet. In: GitHub [Internet]. [cited 24 May 2019]. Available: https://github.com/AlexeyAB/darknet

27. Cartucho. Cartucho/mAP. In: GitHub [Internet]. [cited 27 May 2019]. Available: https://github.com/Cartucho/mAP